Universidad Autónoma de Madrid
Escuela politécnica superior

Doble Grado en Ingeniería Informática y Matemáticas

Trabajo Fin de Grado

ESTUDIO Y ANÁLISIS DE PROPAGACIÓN DE PROGRAMAS MALICIOSOS EN REDES DE COMPUTADORES

Autor: José Luis Suárez Fueyo
Tutor: Franciscoe Borja Rodríguez Ortiz

Julio 2019

ESTUDIO Y ANÁLISIS DE PROPAGACIÓN DE PROGRAMAS MALICIOSOS EN REDES DE COMPUTADORES

Autor: José Luis Suárez Fueyo
Tutor: Francisco de Borja Rodríguez Ortiz

Grupo de Neurocomputación Biológica (GNB
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio 2019

Resumen

Resumen

Tras el desarrollo de Internet a finales del siglo XX, la llegada del siglo XXI ha traído consigo una revolución en la comunicación y mercadotécnia a nivel global. El desarrollo de tecnologías adaptadas a internet permiten que dos personas en lugares opuestos del mundo sean capaces de comunicarse y hacer negocios en cuestión de segundos sin la necesidad de estar físicamente en el mismo lugar. Sin embargo, el desarrollo de nuevos modelos de negocio en internet ha dado lugar a la aparición de nuevos tipos de criminales que usan y desarrollan herramientas tecnológicas como los software maliciosos, o *malware*, para conseguir sus objetivos.

Ante la aparición de este nuevo tipo de delicuentes, surge la necesidad imperiosa de entender bien tanto las herramientas que nosotros utilizamos, así como las que ellos utilizan, a fin de evitar y defendernos ante posibles ataques. A pesar de que la concienciación sobre la importancia de la seguridad informática ha ido aumentando de forma continua durante los últimos 20 años, los expertos siguen demandando una mayor inversión en esta área, dada la evolución y la sofisticación que están alcanzando los métodos de ataque.

El presente trabajo busca alcanzar un mayor entendimiento sobre la topología básica de Internet, los distintos modelos de epidemias existentes y el proceso de la propagación del *malware* que permita elaborar estrategias óptimas frente a ataques que utilicen esta red como medio de propagación.

Aunque existen muchos tipos de software malicioso que utilizan Internet para propagarse, el presente trabajo se centra solo en aquellos que presentan una serie de características:

- **Los ataques no son dirigidos**, es decir, el software que estudiaremos no ataca un dispositivo en concreto sino que busca infectar la mayor población de dispositivos posible.
- El método de propagación son **redes cuya distribución del grado de los nodos sigue una ley potencial, al menos asintóticamente**. En el presente trabajo asumiremos que esta red es Internet, pero ha de notarse que varias publicaciones muestran que existen varias redes complejas con esta estructura.

La razón para centrarnos en este tipo de software malicioso no es otra que este tipo de software es el mas dependiente de la red sobre la que se propaga.

Tras una breve introducción que pretende ilustrar los conocimientos necesarios para el correcto seguimiento del presente trabajo, este proyecto se centrara en la descripción y análisis de experimentos que pretenden dar respuestas a preguntas como:

¿Es importante que la infección se inicie en un hub? ¿Influye el tamaño de la red en la velocidad de propagación del malware? ¿Cuán importante es hallar de forma precisa los parámetros reales de Internet a la hora de modelizar la propagación de estos malware? Desde el punto de vista del atacante, ¿merece la pena dotar al *gusano* de inteligencia?

Palabras Clave

Internet, malware, small-World, scale-free, modelos compartimentales, SIR, SIS, SEIR, hub, grafo, nodo, epidemia, igraph.

Abstract

After the development of the Internet at the end of the 20th century, the arrival of the 21st century has brought about a revolution in global communication and marketing. The development of technologies adapted to the Internet allows two people in opposite places of the world to be able to communicate and do business in a matter of seconds without the need to be physically in the same place. However, the development of new business models on the Internet has led to the emergence of new types of criminals who use and develop technological tools such as malicious software, or *malware*, to achieve their objectives.

Against the emergence of this new type of criminals, there is an urgent need to understand well both the tools that we use, as well as those that they use, in order to avoid and defend against possible attacks. Although awareness of the importance of computer security has been increasing steadily over the past 20 years, experts continue to demand greater investment in this area, given the evolution and sophistication that attack methods are achieving.

Present work seeks to achieve a greater understanding of the basic topology of the Internet that allows the development of optimal strategies against attacks that use this network as a means of propagation.

Although there are many types of malicious software that use the Internet to spread, the present work focuses only on those that present a series of characteristics:

- **The attacks are not directed**, that is, the software that we will study does not attack a specific device but seeks to infect the largest population of possible devices.
- The propagation method is **networks whose distribution of the degree of the nodes follows a potential law, at least asymptotically**. In the present work we will assume that this network is Internet, but it should be noted that several publications show that a number of social networks also have this type of topology.

The reason to focus on this type of malicious software is not other than this type of software is the most dependent on the network on which it spreads.

After a brief introduction that aims to illustrate the knowledge necessary for the correct monitoring of this work, this project will focus on the description and analysis of experiments that aim to answer questions such as:

Is it important that the infection starts in a hub? Does the size of the network influence the speed of malware spread? How important is it to find the exact parameters of the Internet when modeling the spread of these malware? From the point of view of the attacker, is it worthwhile to endow the *worm* with intelligence?

Key words

Internet, malware, small-World, scale-free, compartimental models, SIR, SIS, SEIR, hub, graph, node, epidemic, igraph.

Índice general

Índice de Figuras

Índice de Tablas

1 Introducción

1.1. Motivación del proyecto

Con los avances producidos en los ultimos 30 años en términos de tecnología y sociedad, se ha desarrollado una necesidad del uso de Internet en la mayoría de los ámbitos de nuestras vidas. Mucha gente utilizamos los ordenadores para guardar datos que para nosotros son importantes, desde fotos personales a material de trabajo pasando por una gran variedad de datos de distinta índole e importancia. Además, el auge del IoT lleva la red de redes a cada vez más dispositivos. Estos avances que nos facilitan muchas tareas también han promovido el auge del desarrollo de software malicioso [5][6]. Este tipo de software dañino es creado de forma cada vez más especializada y diversa y aprovecha las nuevas redes complejas que nos enlazan para propagarse causando grandes daños y pérdidas.

Es por esta situación que surge la necesidad de comprender mejor las herramientas que usamos de forma habitual. La investigación sobre software malicioso es cada vez más necesaria e impulsada por empresas y estados [7] ante las pérdidas que causan anualmente. Uno de los campos dentro de esta investigación es como se propagan los *worms*. Los *worms*, en español *gusanos* [8], son un tipo de *malware* que se caracteriza por su capacidad de auto-replicación y su independencia de otros códigos para ejecutarse.

Para poder llevar a cabo este tipo de estudio, es necesario conocer otras disciplinas matemáticas y entender bien el funcionamiento de todo el sistema en el que se produce la propagación. En el caso de este trabajo, se intenta alcanzar un conocimiento que permita realizar unas simulaciones sobre un modelo de Internet con las que alcanzar un mayor conocimiento del funcionamiento de Internet y la propagación de virus sobre ella.

1.2. Enfoque y objetivos

El objetivo principal de este trabajo es alcanzar un mejor conocimiento de cómo afecta la estructura de Internet y sus propiedades en las epidemias de *malware* que, eventualmente pero con cierta frecuencia, inundan los medios de noticias sobre los costes y pérdidas que suponen este tipo de ataques.

El enfoque del documento pretende ser didáctico, recopilando primero los conocimientos necesarios asumiendo una base matemática previa y, posteriormente, utilizar estas bases para realizar unas simulaciones que corroboren los datos, en ocasiones contraintuitivos, que podemos esperar del funcionamiento de Internet trás conocer la teoría detrás de la red de redes.

Antes de realizar estos experimentos y cuyas conclusiones son el fin último del trabajo, se han llevado a cabo los siguientes hitos:

- Entender que es el software malicioso, la importancia que tiene en la actualidad, los diferentes tipos de este software que existen y sus propiedades.
- Ver la similitud entre una epidemia vírica en su sentido tradicional y biológico, y una epidemia vírica sobre las nuevas tecnologías.
- Encontrar entre la aleatoridad que aparenta presentar Internet, una estructura no tan aleatoria que responde a ciertas características y que posee unas propiedades que la hacen particular.
- Aprender a utilizar *igraph* [9][10] para analizar redes complejas, entendiendo como red compleja un grafo con características topológicas no triviales.
- Idear una serie de simulaciones que proporcionen información sobre cómo se propaga un software malicioso de tipo *gusano* sobre una red topológicamente análoga a Internet.
- Obtener conclusiones de las simulaciones realizadas.

1.3. Estructura del documento

Con el fin de alcanzar los objetivos propuestos se ha desarrollado la siguiente memoria que está organizada como se indica:

- **Estado del arte**: En este capítulo se presentan una serie de nociones necesarias para el entendimiento de las simulaciones y las conclusiones posteriores. Estas nociones se dividen en tres temas: Malware, Modelización aplicada a la epidemiología y La topología y los modelos de Internet.
- **Análisis y diseño**: En esta sección se resumen las decisiones de análisis y diseño del proyecto. Se empieza enumerando las diferentes etapas que se han seguido para después enumerar una serie de decisiones teóricas sobre el marco de las simulaciones como son el modelo compartimental a utilizar, el diseño de los grafos a utilizar y la caracterización del *malware*.
- **Desarrollo e integración del entorno**: Si en el apartado anterior se discuten las decisiones teóricas tomadas para la realización del presente trabajo, en este apartado se abordarán las decisiones técnicas como el lenguaje de programación utilizado, las librerías necesarias o la visualización de los datos.
- **Simulaciones realizadas y resultados**: Aquí se recopilan distintas las distintas simulaciones donde se intenta imitar el comportamiento de un *gusano* sobre un grafo similar al de Internet asumiendo distintas premisas con el fin de demostrar ciertas hipótesis.
- **Conclusiones y trabajo futuro**: Finalmente, en el último capítulo del documento se discute sobre los resultados obtenidos y las conclusiones alcanzadas, además de enunciar algunas líneas posibles de investigación futuro y posibles retos en relación al tema de estudio del Trabajo de Fin de Grado.

2

Estado del arte

2.1. Introducción

La epidemiología de *malware* sobre redes complejas es una rama de estudio que necesita bases de otras disciplinas de conocimiento como son la modelización, la teoría de grafos y algunas nociones sobre topología. En este capítulo repasaremos brevemente los conceptos necesarios para entender el desarrollo de este proyecto, siguiendo el siguiente esquema:

I. **¿Qué es el *malware*? ¿Qué tipos de malware existen?** Durante la sección 2.2 intentaremos definir que es el software malicioso, o *malware*, cómo influye en nuestra vida y como clasificarlo de acuerdo a ciertas características.

II. **¿Cómo se modela una epidemia? ¿Hay relación entre una epidemia biológica y una virtual?** En la sección 2.3 analizaremos la similitudes entre una epidemia tradicional, es decir, aquella que se produce entre seres vivos y se propaga a través de distintos vectores biológicos; y las epidemias que atañen a este trabajo que son las epidemias de *malware*. En esta sección, también se repasarán algunos modelos clásicos que involucran los siguientes cuatro estados: Susceptibles(S), Infectados (I), Expuestos (E) y Recuperados (R). Estos cuatro estados dan lugar a distintos modelos como el modelo Susceptible-Infectado-Susceptible (SIS) o el modelo Susceptible-Infectado-Recuperado (SIR).

III. **¿Qué es un grafo? ¿Para qué son útiles los grafos en este trabajo?** La mayoría de las personas no se preguntan cuál es la estructura de Internet, sin embargo, este tipo de estructura se puede estudiar como un grafo. Por esto mismo, en la sección 2.4, se presentan algunas definiciones básicas de la Teoría de Grafos y su aplicación en este problema en concreto.

IV. **¿Qué grafo puede representar de forma, lo suficientemente precisa, Internet?** Una vez que sabemos que Internet puede ser representado por un grafo, en la sección 2.5, resumiremos el estado actual de la investigación sobre la representación de Internet como un grafo y discerniremos sobre cuál es el modelo que debemos utilizar en las simulaciones posteriores.

2.2. Malware

El software malicioso o *malware* es, hoy en día, una de las amenazas más significativas en nuestra vida cotidiana. Diferentes sistemas electrónicos que usamos día a día como los teléfonos móviles u otros electrodomésticos son, además de los ordenadores, susceptibles de ser afectados por software malicioso. Aunque intentaremos mostrar una clasificación general más adelante en el Anexo A, es necesario mencionar la amplia variedad de software malicioso tanto en su función como en su implementación o dependencias.

Uno de los tipos de *malware* existentes son los *ransomware*. Este tipo de programa es similar en su implementación a un *gusano* y se replica de forma similar, pero su fin es encriptar los ficheros en el dispositivo infectado y extorsionar a la víctima a cambio de la clave de descifrado. Durante el verano de 2017, el *ransomware* era la herramienta preferida por los cibercriminales y solo *NotPetya* causó perdidas [11] de $200-$300 millones a la empresa danesa de logística, AP Moller-Maersk; $150 millones a la empresa británica alimentaria Mondelez International, $220 millones a la constructora francesa, Saint-Gobain; ente otras muchas empresas y particulares. Informes de algunas empresas de ciberseguridad [12][13] muestran la predilección de los criminales por el ransomware, y en las Tablas 2.1 y 2.2 podemos ver un ranking de *malwares* más detectados según MalwarebytesLABS[13].

2016	vs	2017
Fraud Tool	1	Hijacker
Adware	2	Adware
Hijacker	3	Riskware Tool
Riskware Tool	4	Backdoor
Backdoor	5	Ransomware

Tabla 2.1: Ranking de tipos de *malware* más detectados en empresas.

2016	vs	2017
Fraud Tool	1	Adware
Adware	2	Fraud Tool
Riskware Tool	3	Riskware Tool
Backdoor	4	Backdoor
Hack Tool	5	Hack Tool

Tabla 2.2: Ranking de tipos de *malware* más detectados en particulares

No solo empresas y particulares son susceptibles de verse atacados por estos programas, y los gobiernos también se valen de su uso y sufren sus consencuencias [14][15][16].

Una vez que hemos visto que el malware es un peligro al que casi todos estamos expuestos, habría que aclarar que es el malware. Una definición de malware [17] aceptada comúnmente sería la siguiente:*"Un programa que, introducido en un sistema, normalmente camuflado, con la intención de comprometer la confidencialidad, integridad o disponibilidad de los datos de la víctima, aplicaciones o sistema operativo, o con el propósito de molestar o perturbar a la víctima."* Sin embargo, existe un amplio abanico de programas distintos que pueden ser clasificados según distintas características como el medio que usan para propagarse o las acciones que realizan una vez dentro del sistema infectado.

A continuación, haremos una pequeña clasificación en distintos tipos de malware de acuerdo a las siguientes tres características del *malware* [3] junto a su función:

- **Capacidad de auto-replicación y auto-propagación.** Algunos tipos de *malware* tienen la capacidad de extenderse por sí mismos mientras que otros necesitan de una interacción para cumplir su función.
- El **crecimiento de la población** del *malware* describe la evolución de los sistemas infectados debido a la capacidad de auto-reproducción del malware. Un *malware* que no sea capaz de replicarse por sí mismo siempre tendrá un crecimiento de cero, aunque es posible que un programa malicioso con la capacidad de replicarse también tenga un crecimiento de cero.

- **Autosuficiencia**. Podemos diferenciar de acuerdo a esta característica entre el *malware* cuyo código se encuentra en otros ejecutables de naturaleza no peligrosa y que han sido modificados de forma maliciosa, o aquél que tiene su propia entidad y sus ejecutables, código y servicios no entran en contacto con otra aplicación. Cabe mencionar que no son solo ejecutables las aplicaciones sino que también puede ser un sector de arranque del disco, una página de la memoria cache y cada día surgen nuevos métodos [18].

Ahora que conocemos algunas de las características que definen un malware resumiremos en el Anexo A, mediante la Tabla A.1, los tipos de malware más comúnmente conocidos según su función y características. Cabe mencionar que la mayoría de la literatura existente sobre el tema está escrita en inglés y es por esta razón que usaremos los términos en inglés.

Antes de proceder a clasificar el *malware*, vamos a justificar por qué para este trabajo se decidió estudiar aquellos software maliciosos que tienen capacidad de auto-replica y se propagan por Internet. De acuerdo al Computer Security Institute (CSI) [1], el porcentaje de ataques no dirigidos, tanto en empresas como en particulares, es mucho mayor que el porcentaje de ataques dirigidos.

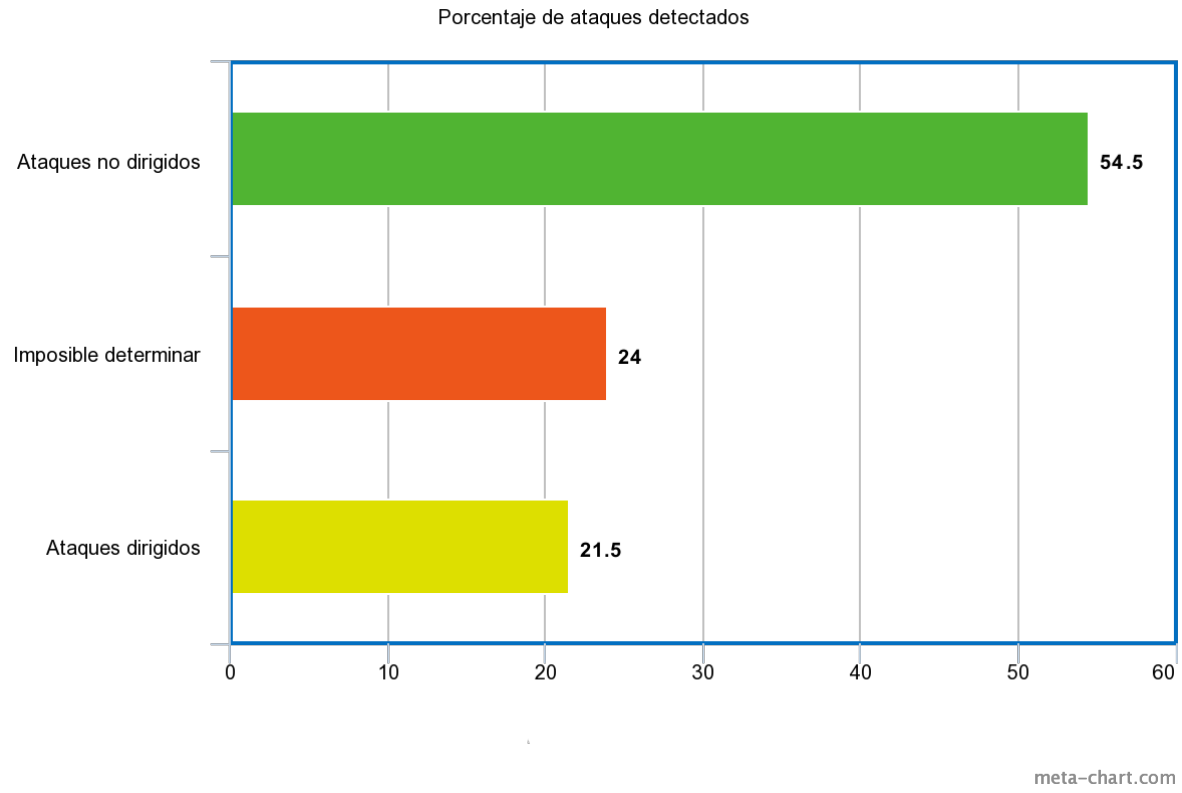


Figura 2.1: Porcentaje de ataques dirigidos vs no dirigidos de acuerdo al CSI. Adaptado de [1]

Además de que este tipo de ataques son los más comunes, otro motivo para centrarnos en ellos es que son los más dependientes de la red sobre la que se propagan. Como se menciona en la Introducción, este trabajo pretende ser en parte didáctico, y al parecer del autor, y lo corrobora la literatura existente sobre el tema, el estudio topológico de Internet es de gran interés. Al estar el autor cursando dos grados, uno de ellos el grado de Matemáticas, también resultó un hecho decisivo para escoger este campo de estudio, el poder utilizar conocimientos ya adquiridos sobre modelización y sobre grafos.

Podemos ver una clasificación del *malware* de acuerdo a las características mencionadas anteriormente en el Anexo A.

2.3. Modelando la epidemiología en redes complejas

La modelización matemática es una disciplina cuyo estudio se centra en expresar, en un lenguaje formal y matemático un fenómeno del mundo real. Esto nos permite teorizar sobre estos fenómenos y predecir resultados matemáticos consiguiendo un mejor entendimiento del mundo real [25][26]. Este proceso se podría simplificar al diagrama de la Figura 2.2.

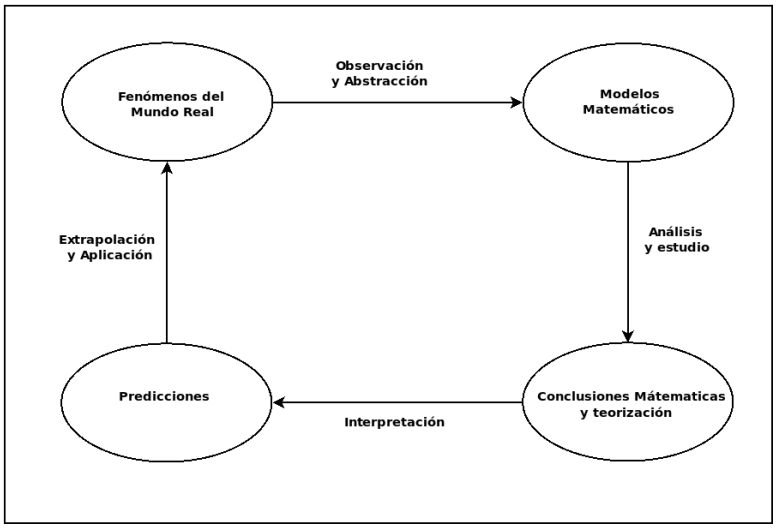


Figura 2.2: Diagrama del proceso de *Modelización*

La modelización nos permite, en casos donde la experimentación puede ser muy costosa o inviable, elaborar hipótesis que posteriormente podemos contrastar con datos empíricos. Es, sin duda, una herramienta muy útil en la investigación moderna y, en el caso de este texto, necesaria ante la imposibilidad de conseguir un mapa de Internet a pesar de los intentos y avances de algunas empresas como Facebook o LinkedIn [27][28][29]. De hecho, la modelización es la mejor opción a la hora de intentar predecir y ganar conocimiento sobre como se va a propagar un gusano o algún otro *malware* que se propagué de forma indiscriminada por la red de redes, donde asumiremos algunas variables, parámetros y relaciones funcionales entre ellos, que no tenemos las capacidades de calcular pero sí de predecir. Antes de entrar a diferenciar los distintos tipos de modelos que existen en la disciplina, y los utilizados en este tipo de problemas, me gustaría citar a un prestigioso matemático en el campo de la modelización, John L. Casti:

"What distinguishes a mathematical model from, say, a poem, a song, a portrait or any other kind of "model", is that the mathematical model is an image or picture of reality painted with logical symbols instead of with words, sounds or watercolors." - John L. Casti

En el caso de este trabajo, la Figura 2.2 no está del todo completa ya que en el proceso de obtener modelos matemáticos existen dos partes. Una primera parte donde se intenta abstraer el fenómeno real con un modelo matemático, mediante la formulación de las ecuaciones que describen la dinámica del proceso, y una segunda que nos permite pasar este modelo a un modelo computacional, es decir, un modelo que podamos probar con los recursos disponibles. Este modelo será el que nos permitirá en el Capítulo 5, realizar distintas simulaciones y extraer algunos resultados y conclusiones interesantes. Atendiendo a algunas características básicas de los modelos, existen distintas categorizaciones, entre las más comunes se encuentran:

- **Modelos estocásticos vs modelos deterministas [26][30].** Los modelos deterministas son aquellos cuyas variables y parámetros son conocidos y no aleatorios, es decir, no siguen una distribución probabilística. En cambio, los modelos estocásticos, también ampliamente

utilizados sí que siguen una distribución probabilística y por tanto sus variables y parámetros en vez de ser un valor, son un rango. Esto hace que el resultado y/o conclusiones también sean un rango[31].

- **Modelos globales vs modelos individuales [30][32].** A la hora de modelar un sistema complejo se pueden adoptar dos posiciones. La primera consiste en pensar en el sistema complejo como una caja negra donde nuestro interés recae en la entrada y la salida con la que trabaja el sistema y donde los componentes de este no son objeto de estudio; o una segunda posición que consiste en modelar el comportamiento individual de cada componente del sistema, siendo este tipo de modelo, por lo general, más extenso y complejo donde cada componente cuenta con una dinámica propia y el conjunto de todas estás dinámicas terminan ofreciéndonos la dinámica general del sistema. Estos dos tipos de modelos nos dan un nivel de detalle muy distinto, y se pueden categorizar también como **modelos con alto nivel de detalle vs modelos con bajo nivel de detalle** Esta categorización de los modelos nos lleva directamente a la tercera mencionada en este apartado; puesto que, normalmente, los modelos globales se basan en ecuaciones diferenciales que dependen del tiempo frente a los modelos individuales que se ejecutan en unos conjuntos de espacio y tiempo discretos.
- **Modelos continuos vs modelos discretos [30][32].** Como deriva de la clasificación anterior, podemos observar que los modelos continuos son aquellos en los que las variables pueden tomar un conjunto infinito de valores mientras que, en los modelos discretos, como su nombre indica, las variables solo pueden tomar cierto número de valores.

No existe un tipo de modelo mejor que el resto y la necesidad de utilizar uno u otro recae en el fenómeno real sobre el que se modeliza, el nivel de detalle que se requiere o si necesitamos modelarlo de forma continua o discreta. Las características mencionadas arriba son algunas de las más importantes a la hora de decidir que modelo es el que más se adecúa, y aun así, ha de **contrastarse mediante la experimentación** para comprobar que el modelo es el adecuado. Hay veces donde es fácil ver que característica cumple un fenómeno o no, como, por ejemplo, una partida de ajedrez tiene turnos finitos y es modelada por un modelo determinista mientras que una epidemia transcurre en un tiempo continuo. La producción de una serie de fábricas puede modelarse a nivel global para cada fábrica mientras que una red neuronal necesita un nivel mayor de detalle. Incluso depende de qué parte del fenómeno, o sistema, se quiera predecir, puede que un mismo fenómeno acepte dos modelos distintos y ambos sean útiles.

Una vez que nos hemos introducido en la disciplina de la modelización, vamos a proceder en la Sección 2.3.1 a presentar algunos de los modelos habituales en la epidemiología y cuál es el que usaremos en nuestras simulaciones del Capítulo 5.

2.3.1. Modelos utilizados en la epidemiología de malware

A lo largo de la historia, el estudio de la propagación de enfermedades en las sociedades ha sido de máxima importancia a la hora de plantear medidas de prevención y minimizar el impacto de enfermedades. Es por razones como estas que nace una disciplina conocida como epidemiologia matemática. Si intentamos hallar los primeros trabajos en este campo, encontramos que ya en 1760 Daniel Bernoulli [33], ante una epidemia de viruela, realizó un estudio sobre qué impacto tendría y como debería hacerse una campaña de vacunación. Desde entonces, muchos otros académicos han planteado modelos y estudiado su adecuación a la realidad.

En este tipo de estudios, los que analizan las epidemias, los modelos más repetidos en la literatura existente son los conocidos como modelos compartimentales [34][35][36], modelos en los que la población se divide en distintos estados o fases conocidos como compartimentos.

Es lógico pensar que ante un virus biológico la población pasa por distintas fases. La clasificación en diferentes estados y las transiciones entre ellos dan lugar a un amplio abanico de modelos distintos. Pongamos como ejemplo una enfermedad conocida por todos, la gripe. Suponemos que durante la época de gripe todo el mundo es susceptible de contraer esta enfermedad, por lo que primero una persona es susceptible de contraer el virus si tiene una interacción con un portador. En el caso de que se infecte, este sujeto de ejemplo ya sería portador de la enfermedad y podría infectar a su vez a otras personas. Más tarde, la persona pasa de incubación a mostrar síntomas, y está pasando por otro estado de la enfermedad distinto, donde ya no puede infectar a otra gente. Finalmente, trás un período de tiempo, la persona se recupera y en el caso de la gripe estaría inmunizado pues ya ha pasado ese virus. En este ejemplo, podemos apreciar cuatro estados distintos y las funciones de transferencia entre ambos.

Figura 2.3: Proceso de la gripe utilizando FirefoxEmoji [2]

Es sencillo ver una relación entre una epidemia de un virus biológico y la propagación de algunos tipos de malware como pueden ser los *gusanos* [37]. Como hemos explicado en el apartado anterior, los gusanos tienden, por regla general, a expandirse a través de algun tipo de red (social, telecomunicaciones, lista de contactos...) de forma indiscriminada y con el fin de infectar al mayor número de ordenadores posibles. Los virus biológicos no son muy distintos pues no atacan a una persona en concreto, sino que intentan propagarse en amplio espectro. A continuación, repasaremos brevemente algunos de los modelos compartimentales clásicos, así como sus diagramas de flujo y las ecuaciones que modelizan sus evoluciones.

2.3.2. Parámetros de los modelos

Las variables utilizadas en las ecuaciones que definen los modelos son las siguientes:

- N – Población total de dispositivos en el sistema. En estos casos tomaremos esta población como constante.
- S – Dispositivos sanos pero susceptibles a contraer la enfermedad.
- E – Dispositivos en periodo de incubación.
- R – Dispositivos recuperados.
- β – Probabilidad de un dispositivo sano de infectarse al exponerse, es decir, tener como vecino de primer grado a un dispositivo infectado.
- $1 / \gamma$ – Promedio de tiempo que un dispositivo está infectado.
- $1 / \epsilon$ – Promedio de incubación para un dispositivo.

2.3.3. SIS con población constante

Como vemos en la Figura 2.4 este modelo divide la población en dos grupos, los susceptibles de infectarse (S) y los infectados (I). Un miembro de los susceptibles pasa a infectados al contraer

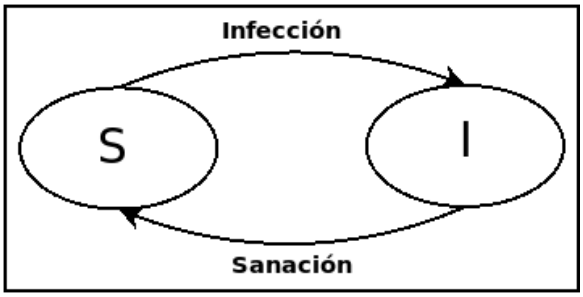


Figura 2.4: Distribución de flujo del modelo *SIS*

la enfermedad, y un infectado vuelve a ser susceptible al sanarse, tomando como axioma que la inmunidad es imposible. La tasa de transferencia en cada unidad de tiempo se razona de la forma siguiente es la siguiente:

- En cada unidad de tiempo existe una población de infectados que intenta propagar la epidemia a una población de susceptibles con probabilidad β. Por otra parte, parte de la población de infectados se cura con probabilidad γ y vuelven a ser susceptibles.
- En contraposición con el compartimento de susceptibles, el de infectados realiza la función inversa. Con probabilidad β los infectados aumentarán en número al entrar en contacto con los suspectibles y disminuirán en proporción γ al recuperarse.

Este proceso puede expresarse con el siguiente sistema de ecuaciones diferenciales ordinarias, asumiendo que la población(N) es constante [38]:

$$\frac{dS}{dt} = -\beta SI + \gamma I \qquad \frac{dI}{dt} = \beta SI - \gamma I \qquad N = S(t) + I(t)$$

2.3.4. SIR con población constante

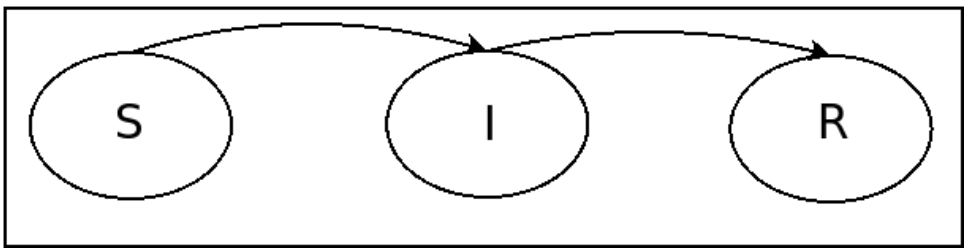


Figura 2.5: Distribución de flujo del modelo *SIR*

Este modelo es la evolución natural del anterior. Sabemos que algunas enfermedades como la varicela, cuyos síntomas solo pueden pasarse una vez, por lo que es lógico pensar que la población ha de dividirse ahora en tres grupos: susceptibles de contagio(S), infectados(I) y recuperados(R). Si como en el apartado anterior, volvemos a asumir una población constante(P), las ecuaciones diferenciales que definen el modelo son [39]:

$$\frac{dS}{dt} = -\beta SI \qquad \frac{dI}{dt} = \beta SI - \gamma I \qquad \frac{dR}{dt} = \gamma I \qquad N = S(t) + I(t) + R(t)$$

Hay que tener en cuenta, que éste modelo que será el utilizado posteriormente en las simulaciones, tiende, asintóticamente al menos, a recuperarse entero; por lo que con el tiempo suficiente toda la población pertenecerá al compartimento de los recuperados.

2.3.5. SEIR con población constante

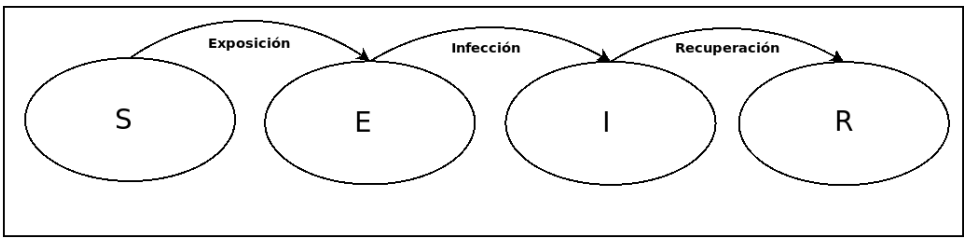


Figura 2.6: Distribución de flujo del modelo *SEIR*

El último modelo clásico que repasaremos en esta sección divide la población en cuatro grupos compartimentales. En algunas enfermedades, el paciente pasa por cuatro fases distintas. Este modelo es útil en aquellos casos en los que una persona, antes de ser infecciosa pasa por un periodo de incubación. Por esto, añade un grupo de individuos infectados que aun no son contagiosos(E). Este modelo viene dado por las siguientes ecuaciones diferenciales[40]:

$$\frac{dS}{dt} = -\beta SI \qquad \frac{dE}{dt} = \beta SI - \epsilon E \qquad \frac{dI}{dt} = \epsilon E - \gamma I \qquad \frac{dR}{dt} = \gamma I \qquad N = S(t) + E(t) + I(t) + R(t)$$

2.4. Introducción a la teoría de grafos

La teoría de grafos [41][42] es una rama de las matemáticas que estudia las propiedades de los grafos. La teoría de grafos tiene sus fundamentos en las matemáticas discretas y en las matemáticas aplicadas. Parece obvio entonces, que debemos empezar definiendo que es un grafo:

*Un **grafo G** es un conjunto no vacío V, de vértices o nodos, y un conjunto A, de aristas, extraído de la colección de subconjuntos de dos elementos de V.*

Los **grafos** son el objeto de estudio de la teoría de grafos, y vemos que su definición involucra dos conceptos nuevos, **vértices** y **aristas**. Aunque existen grafos que no se pueden representar en dos dimensiones, para este apartado asumiremos que sí se pueden para simplificar las definiciones:

*Un **vértice** o **nodo** es un punto en $\mathbb{R}^2$.*

*Una **arista** o **eje** es una dupla de dos nodos, es decir, E = (U, V) con U, V $\in \mathbb{R}^2$.*

Pensando en el trabajo que nos concierne, podemos entender los nodos como todos los dispositivos conectados a la red compleja que vamos a estudiar que, como siempre a lo largo de este documento, suponemos Internet. Las aristas serán entonces las conexiones físicas o cables existentes entre dos dispositivos. Una clasificación muy común y general de los grafos es dividirlos entre grafos dirigidos y grafos sin dirección:

*Diremos que un grafo **es dirigido** cuando para dos nodos U y V, la existencia de una arista E = (U, V) implica que U tiene un enlace con V pero no que V tenga un enlace con U. Es decir, en estos grafos las aristas son duplas ordenadas.*

Aunque esta definición es esencial en la teoría de grafos, no es aplicable a nuestro caso pues supondremos que los enlaces de Internet son bidireccionales. Otra propiedad inherente y de interés de estudio sobre los nodos es el número de enlaces que tiene cada nodo:

*El **grado** de un nodo U, deg(U), es el número de aristas incidentes en ese nodo. En nuestro caso, al tratar grafos sin dirección, el grado de nodo es el número de enlaces que tiene ese nodo.*

Si un nodo tiene grado cero, diremos que es un **nodo aislado**. En las simulaciones realizadas, como cuando un nodo es aislado no tiene sentido considerar que pertenezca a Internet, hemos

modificado el modelo de Albert-Läzlö Barabasi [43] que si permite la existencia de nodos aislados para que no existan estos nodos. Podemos ver este modelo en la sección 2.5.4. Para terminar con esta sección, introduciremos dos nociones sobre la robustez de la red más de relevancia en este proyecto:

*Un **camino** entre dos vértices U y V, es una secuencia de aristas que empiezan en U y terminan en V.* La longitud de un camino viene dada por el número de aristas en el camino. La **longitud media de los caminos más cortos** es determinante a la hora de entender las redes complejas que analizaremos. Pese a la inmensa cantidad de nodos existentes en una red compleja como Internet, la longitud del camino más bajo entre dos nodos cualesquiera de la red, es mucho menor de lo esperado, y esto influye en la propagación de un virus.

La **longitud promedio del camino** se calcula mediante la siguiente formula siendo n el número de vértices del grafo y $d(v_i, v_j$ la distancia entre los nodos i y j:

$$l_G = \frac{1}{n*(n-1)} * \sum_{i \neq j} d(v_i, v_j)$$

La última noción que necesitaremos durante este trabajo sobre la teoría de grafos es el **coeficiente de agrupamiento**. Esta métrica es de gran importancia en nuestro trabajo [44] pues como veremos, algunas redes complejas, y en particular Internet; muestra un alto coeficiente de agrupamiento o *clustering*. Esta medida cuantifica [45] cuán agrupado esta un vértice dentro del grafo, y un alto valor de esta medida facilita la creación de **vecindarios**, es decir, conjuntos de nodos muy agrupados entre si. Se puede hablar tanto de **coeficiente de agrupamiento** o **clustering** a nivel local como global:

- **Clustering global**. Mide la relación entre la cantidad cantidad de triángulos cerrados del nodo frente al total.

  $$C = \frac{numerodetringuloscerrados}{numerodetringulostotales}$$

  Siendo un triángulo, tres nodos conectados mediante dos (abierto) o tres (cerrado) enlaces sin dirección.

- **Clustering local**. Esta medida fue introducida por Watts [44] y calcula cómo de cerca estan un vértice y sus vecinos de ser un grafo completo. Se calcula mediante la siguiente fórmula:

  $$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}}{k_i(k_i - 1)}$$

2.4.1. Ejemplo de grafo.

Para ejemplificar los conceptos básicos del apartado anterior, utilizaremos el grafo de la Figura 2.7:

Recordemos que un **grafo** es un par de conjuntos, siendo uno de estos conjuntos V con los vértices y otro E con las aristas. En el caso de la Figura 2.7, V = {A, B, C, D, E} y E = {(A, C), (A, D), (C, D), (C, E)}. En el ejemplo, al igual que en los grafos de estudio durante este documento, estamos ante un grafo sin dirección.

Un **camino** entre A y E sería l = {(A, C), (C, E)}, cuya longitud sería dos por involucrar dos aristas.

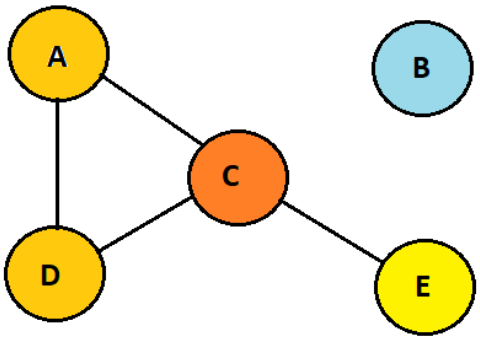


Figura 2.7: Grafo de ejemplo.

Un concepto no introducido en el apartado anterior y relevante a la hora de entender Internet, son los ***hubs***. Estos nodos diferencian las redes aleatorias de las redes *scale free*. La existencia de *hubs* en las redes libres de escala las diferencia notablemente de las redes aleatorias. Entendemos como un *hub* aquél vértice de un grafo cuyo grado es significativamente superior a la media de todos los nodos del grafo.

Veamos ahora cuales en la Tabla 2.3, son los **grados** de los nodos. Los nodos del grafo tienen distinto color según su grado, así será más intuitivo ver el grado de cada nodo.

Nodo	Grado	Aislado/Hub	Aristas
A	2	No	{(A, C), (A, D)}
B	0	Aislado	∅
C	3	Hub	{(A, C), (C, D), (C, E)}
D	2	No	{(A, D), (C, D)}
E	1	No	{(C, E)}

Tabla 2.3: Grado de los nodos del grafo en la Figura 2.7

2.5. La topología de Internet

Los grafos son un concepto muy importante a la hora de entender la realidad. Al ser los grafos un tipo de 'mapa' que describe las relaciones entre nodos, podemos hallar grafos en redes biológicas como las redes neuronales [46], en redes sociales como la red de actores [47] y en otras redes que representan tecnologías como Internet [43][48]. Sin embargo, estas redes no son fáciles de entender; más bien todo lo contrario. A pesar de conocer su existencia estas redes han resultado ser, durante décadas, muy complejas para que las entendiesemos.

Podemos encontrar los primeros modelos en este campo, en los presentados por Pal Erdos y Alfred Renyi en 1959 [49]. Estos matemáticos presentaron un modelo que asumía que todas estas redes complejas se basaban en conexiones aleatorias. Esta teoría fue aceptada en disciplinas tan distintas como la informática, la biología o la sociología. Según se avanzaba en la investigación del campo, surgían nuevas preguntas como la hipótesis de los ***"Seis grados de separación"*** [50], que a pesar de haber sido introducida a finales de los años 20 del siglo pasado, no fue hasta 1998 que se representó con un modelo matemático este fenómeno aparentemente sociológico [44].

Con este nuevo planteamiento, se empezaron a observar redes similares en lugares que no tenían ninguna relación aparente como, por ejemplo, la red neuronal del gusano C. Elegans [51] o la red eléctrica de la costa oeste de Estados Unidos [52].

Una vez que se empezó a comprender que las estructuras de estas redes influían de forma directa en algunos fenómenos observados se empezaron a adaptar los modelos de forma que pudiesen explicarlos todos. Esto permitió que la modelización de las redes pasase de ser un problema propio de una disciplina concreta a un paradigma común entre varias ramas del conocimiento. Además del ya mencionado lema de los ***"Seis grados de separación"*** [50], otro fenómeno llamó la atención de los investigadores. El fenómeno en cuestión es el conocido como ***"rich-get-richer"*** [53], que hace referencia a la mayor probabilidad de recableado a un nodo con un alto grado frente a uno de menor grado. Este fenómeno observable en redes reales permitía explicar la existencia de los *hubs*, nodos de alto grado, que influyen de forma notable en el funcionamiento de Internet.

Trás este breve repaso de la importancia de los grafos en el funcionamiento de Internet, en las secciones siguientes repasaremos dos de los modelos más notorios de este tipo de redes. El primero es el modelo Watts-Strogatz. Este modelo consigue explicar la alta resilencia de Internet además de las comunidades que se crean en este tipo de redes, sin embargo, no consigue explicar el nacimiento de los *hubs* y considera estas redes complejas como redes constantes donde el número de nodos no varía. Una vez explicadas las características de este modelo, procederemos a explicar el algoritmo de cosntrucción propuesto por sus autores. El otro modelo que trataremos será el creado por Albert Laslo Barabasi en 2002 [54]. Mientras que el modelo Watts-Strogatz se publicó en 1998, este modelo es posterior y encuentra explicaciones a algunas incógnitas presentes en el modelo de Watts. Finalmente, se mostrará el algoritmo de construcción propuesto por el autor.

2.5.1. *Small Worlds*

Podemos encontrar uno de los primeros trabajos donde se busca la topología de Internet en el autor D.J. Watts[48] quién inspirado en el famoso lema, *Six degrees of separation*, empezó a observar relaciones interesantes entre la longitud media de los caminos de un grafo y su coeficiente de clusterización en su tésis doctoral junto a su tutor Steven Strogatz. Partiendo de una primera idea de que Internet es similar a un grafo aleatorio de Erdos-Renyi, Watts encuentra ciertas propiedades que le permite definir el fenómeno *small world*. Si es verdad que los primeros modelos de Internet se representaron mediante la generación de grafos con el modelo de Erdos-Renyi y que eran capaces de modelar ciertos comportamientos de Internet, D.J. Watts define una nueva topología de grafos que cumple ciertas cualidades y define mejor algunas redes complejas como Internet. En concreto las redes *small world* se fundamentan en dos propiedades:

- **Alto coeficiente de clusterización.** El autor observa que Internet, como otras redes small-world, se agrupa en una especie de barrios, es decir, pequeños subgrafos donde la mayoría de nodos pertenecientes al subgrafo están conectados entre sí. El significado intuitivo de esta propiedad es el siguiente: Dados dos vértices cualesquiera pertenecientes a la red, si estos grafos no están conectados directamente existe una gran probabilidad de que lo estén de forma indirecta.

- **Media de camino más corto baja.** Dos nodos cualesquiera de la red están conectados por un camino de longitud relativamente baja. Es esta propiedad la que mas ha trascendido a la cultura general y es el conocido fenómeno *small world*. Alguna vez hemos escuchado que solo estamos a seis contactos de distancia de cualquier persona en el mundo, se han escrito una obra y una película sobre el teorema, y existe una página para comprobar esta cualidad en la red social de actores [55]. Aparentemente esta cualidad está presente en muchas otras redes además de Internet, y es por qué la redes sociales, tanto virtuales como físicas, tienen una topología subyacente similar a Internet.

Trás la observación y formulación del modelo, se observó que las redes de mundo pequeño cumplen lo siguiente. Si el grado medio del grafo es k, el número medio de vecinos de primer orden (vecinos inmediatos), será también k, el número medio de vecinos de segundo orden será k^2, el de tercer orden k^3, etc. Esto da lugar a que la distribución de grado de este modelo corresponda a una distribución de Poisson. En la Figura 2.8 vemos un ejemplo de este tipo de grafos generado por *igraph* [10] donde los nodos de mayor grado están coloreados de forma distinta:

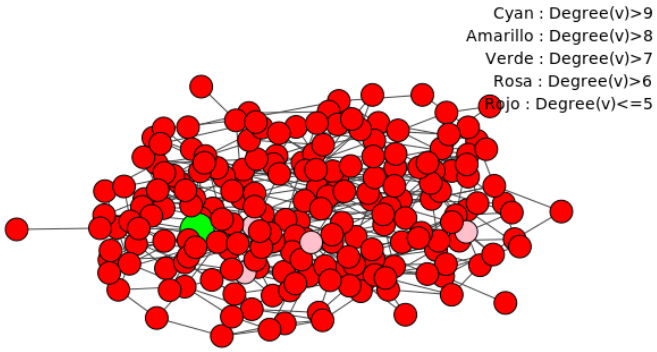


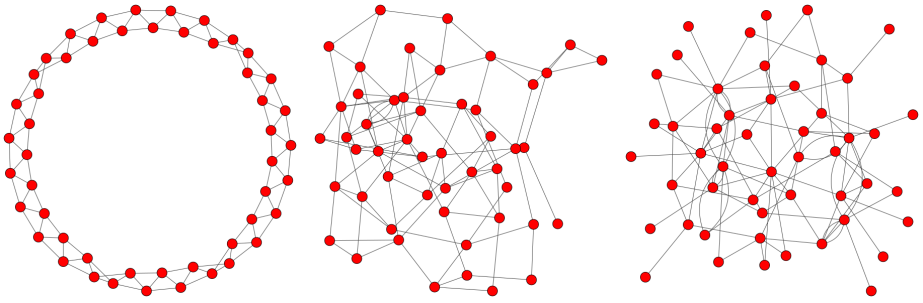
Figura 2.8: Grafo de tipo *Small-world Watts-Strogatz*. N=200

2.5.2. Construcción de un grafo *small-world*

Las redes de mundo pequeño se encuentran entre un grafo regular y aleatorio. Para alcanzar este punto intermedio, el proceso de construcción de la red se detalla a continuación. Lo primero que debemos definir es el tamaño del grafo tanto el número de vértices como el número de aristas.

Partimos de un grafo regular con forma de anillo, que tiene n vértices los cuales están conectados a k vecinos en un inicio. Como hemos mencionado en secciones anteriores, los grafos tratados en este trabajo no tienen dirección.

Escogemos un vértice de forma aleatoria y la arista que lo une con su vecino más próximo. El criterio para escoger este vecino, en el modelo presentado por Watts era el sentido horario. Otro parámetro que define el grafo es p. Este parámetro p define la probabilidad con la que una arista es reconectada a otro vértice aleatoriamente en el grafo, con la condición de que nunca puede haber dos enlaces entre los mismos vértices. En la Figura 2.9 podemos ver grafos para distintos p.



(a) p = 0 (b) p = 0.25 (c) p = 1

Figura 2.9: Redes *small world* para distintos valores de p. Se puede ver la generación en el Código B.4.3.

Seguimos procediendo de la misma forma hasta que hallamos completado una vuelta entera al anillo, rotando de vértice siempre en el sentido horario.

En la siguiente iteración del algoritmo, consideramos los enlaces entre los vértices y sus vecinos de profundidad 2, es decir, los vecinos de sus vecinos. El algoritmo de *recableado* es el mismo que para el paso anterior. Este proceso lo repetimos hasta que hayamos considerado todos los vétices una vez. Como el grado consta de n vértices, cuyo grado inicial es k, el número total de ejes es $|E| = \frac{nk}{2}$, y por tanto serán necesarias $\frac{k}{2}$ iteracciones para completar el algoritmo de construcción. Podemos ver un ejemplo en la Figura 2.10.

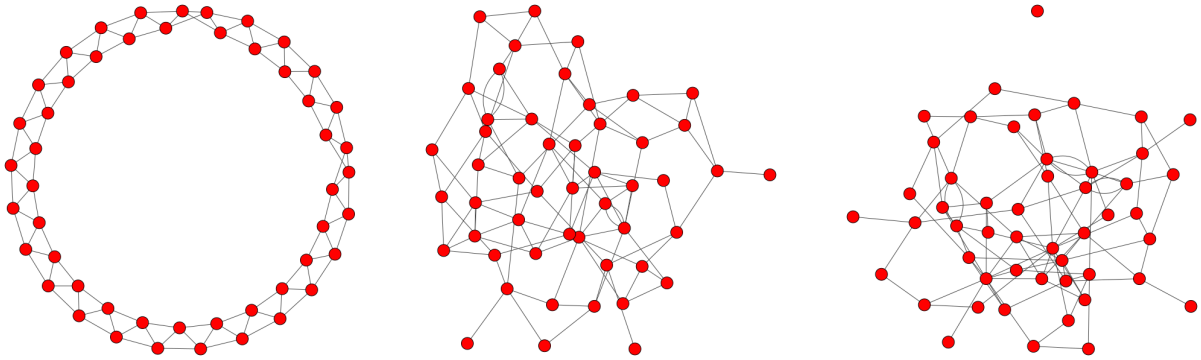


Figura 2.10: Proceso de construcción de red small-world para $p = 0{,}5$. Grafo inicial, primera iteración, segunda iteración. Se puede ver la generación en el Código B.4.4.

2.5.3. Redes *scale-free*

Aunque la investigación de Watts y Strogatz fue sumamente importante y positiva ya que impulsó el estudio de las redes complejas, en 2002 Albert-Laslo Barabasi presentó un nuevo modelo [56] de redes *small world*. Este nuevo modelo reflejaba algunas características de las redes complejas estudiadas por Watts que no estaban representadas en su modelo. Barabasi denominó a estas redes, redes *scale free*.

Una red *scale free* es aquella cuya función de distribución de los grados de los vértices, sigue una ley potencial. Esto quiere decir, que la probabilidad de que un nodo tenga grado k es:

$$P(k) \sim k^{-\gamma}$$

Barabasi observó que este tipo de redes complejas presentan dos características importantes:

- **Red en crecimiento.** Las redes *scale free* están continuamente en crecimiento. Este crecimiento no tiene por qué ser periódico, pero se asume que la red varía el número de nodos con el tiempo y no es constante. Si pensamos en las redes ya mencionadas en las secciones anteriores, como la red de actores, parece obvio que cada día hay nuevos actores y por tanto, en la red, nodos nuevos.
- **Conexión preferencial.** Esta característica quiere decir cuando se crea una nueva conexión o eje entre los nodos de la red, es más probable que los nodos con un grado alto reciban este eje. Esto da lugar a la existencia de los denominados *hubs*, nodos con un grado mucho mayor que la media del grafo. Con este fenómeno, Barabasi consigue modelizar el fenómeno ***rich-get-richer*** mencionado en los apartados anterios. Según el dominio de la red, los *hubs* ganan mayor o menor importancia.

Como en la sección 2.5.1 dedicada a las redes *small world*, terminamos esta sección con un ejemplo generado por *igraph* de un grafo segun el modelo Barabasi-Albert de una red *scale free* en la Figura 2.11.

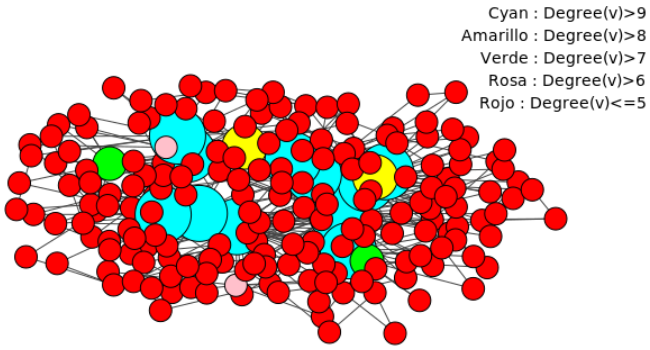


Figura 2.11: Grafo de tipo *Scale free Barabasi-Albert*. N=200

2.5.4. Construcción de un grafo *scale-free*

Además de definir las redes scale-free, Barabasi también propuso un algoritmo de construcción para ellas. Este algoritmo es un simple proceso estocástico basado en un modelo de tiempo discreto en el que cada paso se añade un vértices.

Empezamos con dos vértices y un enlace entre ambos. En los siguientes pasos añadimos un vértice y añadimos ejes entre el nuevo vértice y los antiguos. La probabilidad de que cierto vértice antiguo sea escogido para enlazar con el nuevo es:

$$p_i = \frac{k_i}{\sum_j k_j}$$

siendo p_i la probabilidad de que el nuevo nodo cree un eje con el nodo i. Esta probabilidad da lugar al fenómeno *preferential attachment*.

Otro parámetro que define la red, es el número de enlaces que se crean al añadir cada nodo. Es decir, si cada vez que se añade un vértice al grafo, este se enlaza con uno, dos u otros tres vértices por ejemplo, estas situaciones da lugar a tres redes distintas.

En la Figura 2.12 podemos ver el nacimiento de una red *scale-free* mediante la impresión de los grafos en distintos pasos, T, añadiendo dos aristas en cada paso:

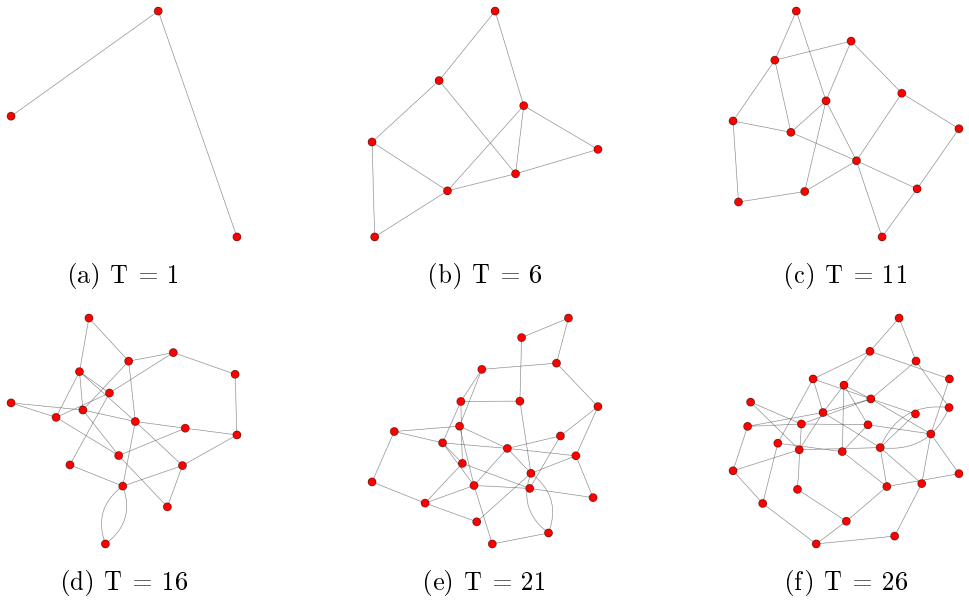
2.6. Herramientas de manejo y análisis de grafos

Para la realización de las simulaciones es necesario la implementación de una herramienta que nos facilite la creación de grafos así como el análisis de los mismos. Existe una amplia variedad de este tipo de herramientas pero puesto que se ha decidido implementar la herramienta en *Python* solo hablaremos de las librerías que tengan soporte en este lenguaje:

2.6.1. Graph-tool

Graph-tool [57] es una librería de *Python* destinada al análisis estadístico de los grafos, es decir, redes. Esta librería destaca porque su núcleo esta implementado en C++ aprovechando la librería *Boost Graph*. Este tipo de implementación permite una mejora notable del rendimiento similar a la que se hubiese obtenido de tratarse de una librería propia de C/C++. Algunas de sus características son:

- Amplias posibilidades para obtener medidas de centralidad, variedad de algoritmos topológicos, manejo I/O de grafos mediante *GraphML*, detención de comunidades...



(a) T = 1 (b) T = 6 (c) T = 11

(d) T = 16 (e) T = 21 (f) T = 26

Figura 2.12: Nacimiento de una red *scale-free* en distintos momentos. Se puede ver la generación en el Código B.4.5.

- Potente capacidad de visualización de grafos.
- Soporte de *OpenMP* para arquitecturas multi-núcleo.

2.6.2. NetworkX

Al igual que *Graph-tool*, *NetworkX* [58] es otro paquete de *Python* destinado al estudio de la estructura, las dinámicas y las funciones de redes complejas. Se trata de una herramienta *open source* y algunas de sus características son las siguientes:

- Cobertura del código del 90 %.
- Diferentes algoritmos y constructores de redes complejas.
- Abstracción de los nodos, permitiendo que distintos tipos de datos como imágenes, texto, ficheros XML, o cualquier otro archivo serializable.

2.6.3. igraph

Finalmente, existe otro conjunto de herramientas de análisis que se centra en la eficiencia y portabilidad, además de contar con una interfaz muy accesible. Esta kit de herramientas, *igraph* [9], es la que utilizaremos durante la elaboración de este proyecto.

Hemos escogido esta herramienta entre todas las posibles, por su amplia documentación [10] y su alta eficiencia que nos han permitido aprender a utilizar la herramienta en un tiempo relativamente corto así como poder realizar las simulaciones en un ordenador personal cuyas modestas prestaciones se reflejan en la Tabla 4.1.

3

Análisis y diseño

3.1. Etapas del proyecto

Para conseguir realizar las simulaciones que se encuentran en la Sección 5, se han seguido una serie de etapas, que pasan desde la documentación en el campo del proyecto hasta decisiones de diseño y alcance del mismo:

I. **Estudio y definición del proyecto**. Ante la idea de entender mejor como se propaga el software malicioso en Internet, fue necesario realizar una lectura de la literatura existente en el campo que permitiése definir el alcance de este proyecto. Mediante la lectura de numerosos artíŋculos centrados en esto mismo [59][60][61][62], se consiguieron definir los siguientes temas que necesariamente se debían estudiar y presentar en el documento antes de la realización de las pruebas:

   a) **Estudio del estado actual de *Malware* y su categorización.** Parece obvio que si pretendemos estudiar la propagación del software malicioso, primero es necesario definir claramente qué es, cuáles son los tipos de software malicioso existentes, quiénes lo utilizan y con qué fines. Por esto mismo, se repasaron algunas de las referencias básicas [63][4] del tema y se elaboró un breve resumen de las nociones básicas en la la Sección 2.2.

   b) **Estudio sobre los modelos de epidemiología actuales.** El problema sobre el que se basa este trabajo es, en esencia, un problema de epidemias o, por sus proporciones, de pandemias. Es, por esto, lógico y necesario repasar los distintos enfoques a la hora de plantear el estudio. Primero, se repasaron los modelos epidemiológicos en su forma más genérica [38][39][40] y después se realizaron lecturas de artículos más enfocados al tema del presente trabajo [26][32][59]. Todo este trabajo se resume en la Sección 2.3.

   c) **Introducción a la teoría de grafos.** Internet, la WWW y otras redes existentes en el mundo real pueden ser representadas como grafos. Representar una red análoga a Internet como un grafo es indispensable para poder realizar un estudio de la propagación del software malicioso sobre esta red. Por esto se repasó la Teoría de Grafos [64] y se recogieron los conceptos necesarios para el correcto entendimiento del trabajo en la la Sección 2.4.

d) **Estudio sobre la topología de Internet y su parametrización.** En los primeros artículos de D.J. Watts [48][44], y en la colección de escritos de Albert-László Barabási [65][56] sobre el análisis de Internet y su estructura, nos enseñan que Internet presenta unas características que podrían influir en la propagación del *malware*. Para entender cuáles son estas características y cómo influyen se leyó información de estos autores entre otros que queda resumida en la Sección 2.5.

e) **Estudio sobre las herramientas para el tratamiento y análisis de grafos.** Finalmente necesitamos una herramienta que nos permita unir todos estos conocimientos para experimentar con ellos y realizar simulaciones. En el caso de este trabajo decidimos utilizar *igraph* puesto que es una librería diseñada para tratar grandes grafos que nos permitan tener una estructura modestamente análoga a Internet. A partir del libro de los desarrolladores de la librería [10], pudimos comprender las posibilidades que nos brindaba *python* junto con *igraph* y que posteriormente nos permitirían comprobar algunas hipótesis surgidas de la investigación.

II. **Elaboración de hipótesis a contrastar.** Trás tener una base de conocimientos sobre el tema, se teorizó sobre cómo se comportaría el fenómeno, la propagación del *malware*, ante distintos sucesos donde la inteligencia del *malware*, la estructura de la red sobre la que se propaga o las funciones de transición entre los distintos estados varían.

III. **Diseño de la herramienta para realizar las simulaciones.** Una vez planteadas las pruebas que se pretenden realizar y establecidas las tecnologías a usar, se implementó un conjunto de clases de Python y funciones que permitiesen realizar las simulaciones diseñadas.

IV. **Desarrollo del código de las simulaciones.** Con las simulaciones definidas y la herramienta implementada, se prepararón las distintas simulaciones recogidas en el el Capítulo 5.

V. **Análisis de los resultado y observaciones sobre estos.** Finalmente, en el Capítulo 6, se resumieron las distintas simulaciones realizadas mediante los enunciados de las hipótesis que impulsaron cada simulación, las premisas de cada una y las tablas y/o gráficas que nos permitiesen visualizar los resultados.

3.2. Requisitos de la herramienta

Para poder realizar el estudio recogido en este trabajo, se ha necesitado desarrollar, utilizando distintas tecnologías, una herramienta que permitiese realizar las simulaciones del Capítulo 5. A continuación se presentan los requisitos funcionales y no funcionales que debe cumplir la herramienta:

3.2.1. Requisitos Funcionales

A continuación se detallan los requisitos funcionales mínimos que ha de cumplir nuestra herramienta de simulación para llevar acabo su propósito. Estos requisitos especifican las funciones que la herramienta debe ser capaz de ejecutar:

- **RF1. Creación de grafos.** La herramienta ha de ser capaz de crear distintos tipos de grafos entre los que están incluidos los modelos Lattice, Watts-Strogatz y Barabasi-Albert.

- **RF2. Manejo de grandes grafos.** La herramienta ha de ser capaz de manejar grafos de grandes dimensiones.
- **RF3. Soporte de Modelos Compartimentales.** La herramienta ha de tener clases y estructuras que soporten los distintos modelos compartimentales como SIS, SIR y SEIR.
- **RF4. Realizar las simulaciones de propagación.** Mediante el uso de los grafos y los sistemas compartimentales, la herramienta ha de ser capaz de simular una propagación de malware.
- **RF5. Capacidad de variar el número de infectados y susceptibles.** Cómo las simulaciones cuentan con distintos escenarios, algunos de ellos varían el porcentaje de susceptibles de la población total o el número inicial de infectados. Esta funcionalidad debe ser soportada por el sistema.
- **RF6. Calibración de los parámetros de la simulación.** La simulación ha de permitir modificar también otros parametros propios de la simulación, como las tasas de infección y recuperación.
- **RF7. Capacidad de guardar y cargar grafos.** La herramienta debe manejar los grafos en un formato que permita su guardado en un fichero y, posteriormente, ha de ser capaz de cargar grafos en este formato desde un fichero.
- **RF8. Capacidad de mostrar los grafos de forma visual.** Para poder mostrar los resultados de las simulaciones, a veces es necesario dibujar el grafo. La herramienta debe ser capaz de mostrarlos para un tamaño aceptable.
- **RF9. Personalización de la visualización.** Los grafos deben tener opciones de personalización como tamaño de los vértices y colores para la fácil comprensión de las imágenes.
- **RF10. Visualización de gráficas con resultados.** Para algunas simulaciones será necesario que los datos númericos obtenidos sean reflejados en una gráfica. La herramienta debe proveer esta función.

3.2.2. Requisitos No Funcionales

Además de los requisitos funcionales mencionados en la sección anterior, también existen otros atributos de calidad que ha de cumplir la herramienta. A continuación, los requisitos no funcionales principales:

- **RNF1. Generación rápida de grafos de tamaño considerable.** La generación de grafos es una función básica de la herramienta. Por ser este un proceso repetido a lo largo de todo el trabajo, es necesario que la herramienta lo haga en un tiempo aceptable.
- **RNF2. El código de la herramienta ha de ser intuitivo y legible.** Como la herramienta no cuenta de una interfaz gráfica, es necesario que el código sea accesible para poder modificar los parámetros necesarios para cada simulación.
- **RNF3. Código compatible con Ubuntu 18.04.** Como el *sistema operativo* dónde se realizarán las simulaciones es Ubuntu 18.04, la herramienta debe ser compatible con este sistema.
- **RNF4. El código debe estar propiamente estructurado y comentado.** Como las modificaciones se realizan sobre el código, es necesario que se organice en modulos distintos y esté comentado.

3.3. Diseño de la infraestructura

Una vez que hemos definido las funcionalidades de nuestra herramienta, tenemos que definir los diferentes agentes y estructuras que intervienen en las simulaciones. Esto pasa desde caracterizar los *malware* que utilizaremos a especificar los parámetros con los que generaremos nuestra red análoga a Internet. Otro apartado relevante de la infraestructura, es el modelo compartimental que usaremos.

3.3.1. Caracterización del malware a propagarse

Uno de los factores clave de las simulaciones del Capítulo 5 es el *malware* cuya propagación vamos a simular.

En nuestro caso, ya hemos mencionado en la introducción, que usaremos un *malware* de tipo *gusano*. Estos software maliciosos se caracterizan por extenderse de forma no dirigida, es decir, no buscan infectar un dispositivo en concreto sino el mayor número de ellos. Esto quiere decir que los límites de propagación vienen dados en gran medida por la estructura sobre la que se propagan que especificaremos en la Sección 3.3.3.

Para definir la tasa de infección y recuperación de la Simulación 5.4 nos basaremos en un caso conocido y reciente como es la epidemia de *WannaCry* [66]. No nos centraremos en las vulnerabilidades que utilizó para infectar los dispositivos, sino que nos centraremos en aquellas características que nos permitan definir nuestro *malware* simulado. Esta epidemia comenzó el 12 de Mayo de 2017 y de ella conocemos los siguientes datos:

- 98 % de las víctimas utilizaban Windows 7.
- 0.07 % de las víctimas pagaron el rescate que les permitíŋa desinfectarse.
- El parche que corrigió las vulnerabilidades fue lanzado el 18 de Mayo.
- La infección llegó a alrededor de 400.000 dispositivos infectados.

Con estos datos y el conocimiento de que en Junio de 2017 la cuota de mercado de Windows 7 era 49.04 % podemos establecer unos parámetros para la simulación correspondiente. En esta simulación, procederemos de la siguiente forma:

- Esta infección duró 7 días antes de ser reparada la vulnerabilidad, por lo que durante esta simulación realizaremos 14 pasos de la infección, simulando estos 7 días y los 7 posteriores.
- Como hemos mencionado antes, la población principalmente afectada han sido los usuarios de Windows 7. Como la cuota de mercado de este S.O. era del 49.04 % asumiremos que nuestra población susceptible inicial será el 50 por ciento del total de la red.
- Durante los 7 primeros días, no existía un parche para la vulnerabilidad y solo el 0.07 % pagaron el rescate. Supondremos que la tasa de recuperación, γ, es 0.0007 durante los primeros días y a falta de datos tras el lanzamiento del parche, se recuperará con una $\gamma = 0{,}5$. Suponemos una γ tan alta ya que al ser la mayoría de infectados usuarios de Windows 7 y este contar con un sistema de actualizaciones automáticas, es fácil llegar a estas cifras.
- El último dato que nos falta por inferir es la velocidad de propagación, β. Como este *malware* se centraba en una vulnerabilidad que compartían todos los susceptibles, podemos asumir que infecta a los susceptibles con $\beta = 1$.

3.3.2. Diseño del modelo compartimental

En este trabajo, se estudian epidemias de *malware*. En particular, estos *malwares* que se estudian en el presente documento, suelen basarse en distintas vulnerabilidades que se descubren puntualmente y los criminales utilizan para sus propios fines. Es por esto que tenemos que adaptar levemente alguno de los modelos presentados en la Sección 2.3.

Si analizamos los casos existentes mencionados en la Sección 2.3, vemos que durante una epidemia de *malware* se sigue el siguiente proceso:

1. **Se descubre una vulnerabilidad.** Las tecnologías están en continua evolución, y el constante lanzamiento de nuevas versiones, nuevos sistemas operativos, nuevas herramientas, etc; provocan que existan nuevas vulnerabilidades cada poco tiempo. A veces, los criminales ni si quieran han de buscar ellos mismos las vulnerabilidades, como en el caso ya mencionado de *Wannacry*, donde la vulnerabilidad había sido destapada por un grupo de investigación dos meses antes.

2. **Los criminales desarrollan el software malicioso.** Este paso es despreciable en cuanto a tiempo se refiere, puesto que cada vez hay mayor profesionalización del cibercriminal, su tiempo de respuesta es muy bajo.

3. **Los criminales infectan al paciente cero.** Con la herramienta ya preparada, los criminales buscan una primera víctima y esperan.

4. **El software malicioso se expande.** Esta parte del proceso es la que simularemos. ¿Cuánto es el daño causado antes de la reparación de la vulnerabilidad? ¿Cómo lo podemos minimizar? La mayor parte de la investigación de las empresas de ciberseguridad se centra en el análisis de los distintos vectores de ataque. Sin embargo, el pensamiento del autor es que un mayor entendimiento de cómo se propaga el *malware*, la estructura sobre la que lo hace y el proceso en sí mismo, reportaría grandes beneficios en materia de seguridad.

5. **La vulnerabilidad se repara.** Finalmente, las empresas, los investigadores y organismos gubernamentales reparan la vulnerabilidad y los dispositivos se actualizan progresivamente si fuese necesario.

Analizando los distintos pasos del proceso de propagación, podemos ver que los dispositivos o nodos de la red tienen cuatro estados posibles:

1. **Nodos inmunes.** Las vulnerabilidades no siempre afectan a todos los dispositivos, por lo tanto habrá nodos en la red que nunca pasan por los demás estados.

2. **Nodos susceptibles.** En contraposición al estado anterior, ante una nueva vulnerabilidad, además de haber nodos inmunes habrá nodos que sufran la vulnerabilidad y por tanto sean susceptibles.

3. **Nodos infectados.** Eventualmente, algunos de los nodos susceptibles, entrarán en contacto con un nodo infectado y se contagiarán del *software malicioso*. Los efectos de la infección son distintos según el tipo de *malware* pero asumiremos que esto no influye en la propagación.

4. **Nodos recuperados.** Finalmente, ante una epidemia de *malware*, hay un momento en el que se repara la vulnerabilidad utilizada tanto para la propagación como para la infección.

En la mayor parte de las simulaciones seguiremos este esquema que corresponde al modelo SIR. Sin embargo, se propone la implementación de un modelo *XSIR* que añade un compartimento para los nodos que nunca han sido ni serán susceptibles de infección. A partir del modelo XSIR, podemos obtener un modelo SIR si ignoramos los vértices en el compartimento X, es decir, el de los inmunes. Sin embargo, en ocasiones es necesario representar la totalidad de la red y estos nodos aunque inmune pueden actuar como medio de propagación o defensa frente al *malware*.

3.3.3. Parametrización y modelización de una red análoga a Internet

Los ordenadores, los móviles, incluso las lavadoras o neveras, son dispositivos susceptibles de ser infectados por un software malicioso. Cada día son más los dispositivos que cuentan con un software que los controla y que están conectados a Internet. Esto conlleva que Internet tenga un gran tamaño que nos impida modelarlo. Es más, actualmente no se ha conseguido cartografiar la red de redes a pesar de las investigaciones e inversiones que se realizan para ello.

Sin embargo, como hemos estudiado en la Sección 2.5 se ha visto que Internet pertenece a un tipo de redes complejas y que comparte características con otras redes. Si recordamos el apartado del estado del arte donde hablabamos sobre la topología de Internet, recordaremos que Internet es una red *scale free*. Estas redes teníŋan una función de distribución del grado de sus nodos del tipo:

$$P(k) \sim k^{-\gamma}$$

Varias investigaciones han demostrado que en el caso de Internet, $\gamma = 2{,}2$ [65][56], y este será el parámetro utilizado durante las simulaciones.

Otro aspecto importante a la hora de preparar el grafo que nos sirva de estructura, es su tamaño. Como ya hemos mencionado al principio de este apartado, el tamaño de Internet es tan colosal que no podemos representarlo. Sin embargo, como veremos en la Sección 5.1, distintas redes con mismo γ y distintos tamaño, se comportan de forma muy similar. Por esto, adaptaremos el tamaño de la red en cada simulación según las necesidades de la misma, puesto que este parámetro no es esencial a la hora de predecir cómo se comporta la propagación del *malware*.

La posibilidad de crear este tipo de grafo con un cierto γ en particular, nos la ofrece la libreríŋa escogida para la realización de la herramienta, *igraph*. Esta libreríŋa cuenta con el siguiente constructor:

```
Barabasi(n, m, outpref=False, directed=False, power=1, zero_appeal=1,
implementation="psumtree", start_from=None)
```

Explicándose los significados de sus parámetros en la Tabla 3.1.

Parámetro	Significado	Valor en las simulaciones
n	Número de vértices	Depende de la simulación
m	Número de aristas que salen de cada vértice	4.16 [67]
outpref	Flag para hacer depender del grado de salida la conexión preferencial	False
power	Constante de potencia	2.2 [43]

Tabla 3.1: Tabla de parámetros del constructor de grafos.

3.4. Inteligencia del *gusano* propuesta

En la simulación de la Sección 5.3, se teoriza sobre un caso donde el *malware* no se propagué indiscriminadamente sino que pase por dos fases:

I **Primera fase: Ataque a *hubs*.** En nuestro supuesto caso, en los primeros cinco pasos, el *malware* se dirige especíŋficamente a por los nodos de mayor grado. Este supuesto no es tan descabellado puesto que dispositivos con funciones similares tienden a tener el mismo sistema operativo. Si los cibercriminales encontrasen un *exploit* en ese sistema operativo, podríŋan diseñar una herramienta mixta donde primero atacase a los *hubs* y después procediese a desplegar otro *malware* que explotase una vulnerabilidad más común.

II **Segunda fase: Ataque no dirigido.** Cómo mencionamos en el párrafo anterior, suponemos que este software malicioso es una herramienta mixta, que trás un cierto periodo de tiempo libera otro *malware* que ataca de forma no discriminada.

4

Desarrollo e integración del entorno

Durante este capítulo se resumirán las herramientas necesarias para la elaboración de este trabajo. Desde los dispositivos utilizados para la realización de la herramienta y las simulaciones hasta las librerías que son necesarias para su correcta implementación y ejecución.

Después se resumirá la organización del código y se mostrará un diagrama de clases del mismo. Este diagrama solo corresponde al código utilizado por las simulaciones puesto que el resto de código elaborado han sido pruebas o clases independientes las cuáles no tiene sentido introducir en el diagrama.

Todas las librerías utilizadas, así como el sistema operativo sobre el que se desarrolló el trabajo se encuentra bajo licencia *open source*. La filosofía *open source* defiende que los productos licenciados bajo la misma han de ser libres y abiertas y que los productos derivados de los mismos han de tener también licencia *open source*. Este tipo de productos impulsa el aprendizaje.

4.1. Entorno de desarrolo y simulación.

Para la realización de este trabajo se utilizaron dos ordenadores personales. Uno para la elaboración del documento en síŋ cuyas prestaciones son irrelevantes y otro para el desarrollo de la herramienta y la ejecución de las simulaciones. Las prestaciones de este segundo ordenador quedan recogidas en la Tabla 4.1.

El código implementado durante este proyecto, ha sido integramente realizado en ***Python*** debido a la existencia de las herramientas necesarias en este lenguaje así como la familiaridad del autor con el mismo.

Además, en la Sección 4.1.2 se enumerarán las librerías y paquetes de Linux necesarios para la réplica de las simulaciones. Recordar al lector que el código de este proyecto esta recogido en el Anexo B.

4.1.1. Sistema hardware

	Ordenador
Procesador	Intel(R) Core(TM) i5 CPU M480@2.67GHz, 2 Núcleos
RAM	4 GB DDR2
SO	Ubuntu 18.04 LTS (Bionic Beaver)

Tabla 4.1: Tabla con las prestaciones del equipo utilizado para el desarrollo y la simulación de este proyecto.

4.1.2. Software necesario.

Como vemos en la Tabla 4.1 este proyecto se ha desarrollado utilizando la última versión estable de Ubuntu disponible. Además de ser este el sistema operativa sobre el que realizaremos la implementación de la herramienta asíŋ como la ejecución de las simulaciones fue necesario la instalación de algunos nuevos paquetes asíŋ como el uso de otros ya instalados previamente. A continuación, se enumerarán los paquetes necesarios para poder realizar las simulaciones:

1. ***build-essential* y *python2.7***. Los primeros paquetes que necesitamos son los básicos para trabajar con *python*. En nuestro caso trabajaremos con la versión 2.7 de *python* por ser la que más habitualmente usa el autor. Los comandos para su instalación son los siguientes:
   sudo apt-get install build-essential
   sudo apt-get install python2.7

2. ***NumPy***. *NumPy* es una de las libreríŋas más usadas en la comunidad cientíŋfica. Se puede encontrar tanto de forma independiente o podemos encontrar sus funcionalidad básica en el paquete *pylab*. En nuestro caso, usaremos la versión independiente que se instala de la siguiente forma:
   sudo apt-get install python-numpy

3. ***random***. Aunque en la Sección 2.5 hemos visto que Internet no es una red aleatoria, si que existe cierta componente aleatoria en su creación. Lo mismo sucede con la propagación de *malware* que se basa en procesos estocásticos. Para replicar esta aleatoriedad utilizaremos la libreríŋa *random*. Aunque hayamos enumerado esta libreríŋa puesto que es una de las que usamos, este modulo ya viene instalado en la funcionalidad básica de *Python*.

4. ***iGraph***. Finalmente necesitamos usar la libreríŋa especíŋfica para el alcande de este documento, que hemos decidido que sea *igraph*. Para ello basta con instalarla con el siguiente comando:
   sudo apt-get install python-igraph

4.2. Distribución del código

Para la realización de este proyecto, se han implementado una amplia variedad de clases. Algunas de estas clases no son utilizadas en las simulaciones pero fueron necesarios para seguir un proceso de aprendizaje correcto sobre la epidemiología del *malware*.

Existen una serie de clases que no tienen dependencias con otras. estas clases son las utilizadas para la generación de las imágenes que se pueden encontrar en la sección de construcción de grafos small world y grafos scale free.

Después existe una serie de clases relacionados de la forma mostrada en la Figura 4.1 para realizar las simulaciones del proyecto. Aquí podemos diferenciar clases en distintos niveles: las simulaciones, funciones del núcleo, modelos epidemiológicos y una clase para el modelo compartimental y otra para los compartimentos.

4.2.1. Diagramas UML de las Simulaciónes

En esta sección se presenta el diagrama de clases de las simulaciones. Como podemos ver en el Anexo B estas no son las únicas clases desarrolladas para el proyecto, sin embargo, los otros códigos constan de un solo archivo independiente del resto por lo que se ha decidido no representarlos en el diagrama de clases de la Figura 4.1.

Recordar al lector, que todo el código que se menciona durante este documento puede ser encontrado en el Anexo B.

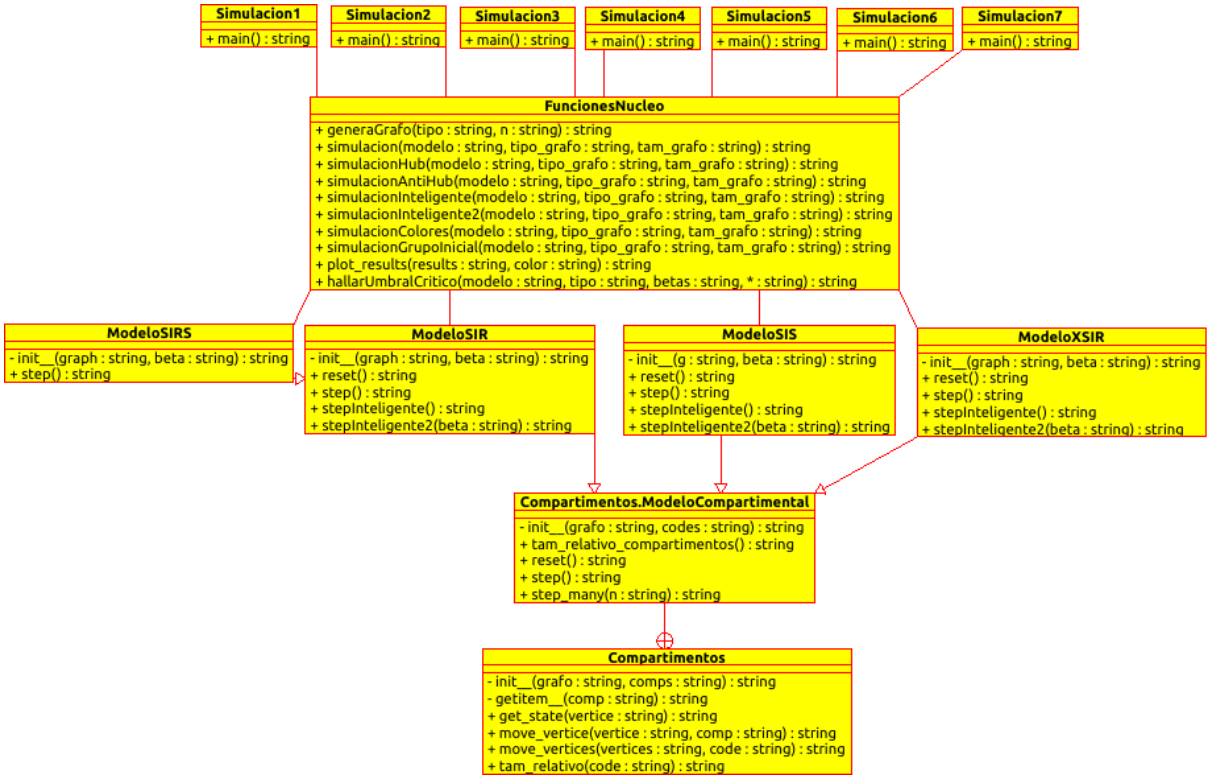


Figura 4.1: Diagrama de clases con la distribución del código de las simulaciones del proyecto. Se puede ver el código en la Sección B.1.

5

Simulaciones realizadas y resultados

Durante esta sección experimentaremos con el fin de entender más de ciertas hipótesis que han surgido durante la investigación realizada. En esta sección se realizarán simulaciones para responder a algunas preguntar que al autor le han parecido de interés. Estas preguntas son:

I. **¿Influye el tamaño de la red en la velocidad relativa de propagación?** Es obvio que el tamaño de la red influye en el número total de infectados durante una epidemia de *malware*. Sin embargo, algo no tan obvio es si el tamaño influye en la proporción de población infectada. A parte del interés de esta característica en sí misma, esta prueba nos es necesaria para después poder interpolar los resultados de nuestra simulación sobre un modesto grafo a una red mayor como Internet.

II. **¿Cuánta importancia tienen los paciente cero en una epidemia de *malware*?** Las propiedades de una red libre de escala como Internet imponen que existan nodos con un grado muy superior al resto. La lógica nos hace pensar que comenzar en estos nodos de alta conectividad haría que la epidemia tuviese mayores consecuencias pero esto puede que no sea cierto. En la segunda simulación intentamos ver la importancia que tiene empezar una epidemia en los *hubs*.

III. **¿Es eficiente diseñar un *malware* inteligente para causar una epidemia?** Para poder entender bien como enfrentarnos en la epidemia, es necesario entender como piensan los criminales detrás de ella. Esto nos lleva al siguiente planteamiento trás la prueba anterior: Un ciber delincuente podría diseñar un *malware* que tuviese una fase inicial donde atacaría de forma inteligente y certera a los hubs más cercanos desde el *paciente cero*. Aunque este tipo de programa sería muy complicado de llevar, ¿cuáles serían sus consecuencias? Este escenario es el que planteamos durante la tercera simulación.

IV. La cuarta simulación es la más visual de todos e intenta **replicar el proceso seguido por *WannaCry***. Esta simulación cuenta con la red más pequeña de todas a fin de que sea visual.

V. La siguiente simulación intenta evaluar la importancia de que **el grupo de infectados iniciales sean vecinos o no**. Para ello se realizan dos simulaciones con tres infectados iniciales cada uno. En la primera se fuerza a que sean vecinos y en la segunda se utilizan el algoritmo de simulación habitual.

VI. La última simulación del trabajo busca responder a cómo se comportará la propagación dado un β **fijo y un** γ **variable**.

5.1. Relación entre tamaño de la red y la propagación

La primera simulación que se ha decidido realizar era necesaria para que el estudio realizado en este proyecto tuviese sentido. Como no es posible representar Internet en su totalidad se necesitaba comprobar si la propagación de *malware* sobre redes *scale free* se comportaba de forma similar para distintos tamaños de la red. Para ello, se ha decidido realizar distintas simulaciones de la propagación de *malware* con la misma parametrización salvo el tamaño de la red. Podemos ver los parámetros de la simulación en la Tabla 5.1.

# nodos	F. de propagación	Modelo	Ejecs.	Tiempo	β	γ	Infectados	Susceptibles
10^2	simulacion	SIR	10	100	0.025	0.01	1	99
10^3	simulacion	SIR	10	100	0.025	0.01	1	999
$5 * 10^3$	simulacion	SIR	10	100	0.025	0.01	1	4999
10^4	simulacion	SIR	10	100	0.025	0.01	1	9999
$2{,}5 * 10^4$	simulacion	SIR	10	100	0.025	0.01	1	24999
$5 * 10^4$	simulacion	SIR	10	100	0.025	0.01	1	49999
10^5	simulacion	SIR	10	100	0.025	0.01	1	99999
10^6	simulacion	SIR	10	100	0.025	0.01	1	999999

Tabla 5.1: Tabla con los parámetros de la Simulación 1.

Como vemos en la Figura 5.1, aunque el tamaño de la red marca la proporción máxima de infectados, las gráficas para todas las simulaciones tienen una tendencia similar. Esta gráfica presenta la proporcion de infectados a lo largo del tiempo, siendo la proporción de infectados el promedio de diez ejecuciones para cada simulación. Si el número de ejecuciones fuese mayor, las gráficas se diferenciarían sin cortarse, pero por limitaciones de HW, esto no es posible. Esto nos permite realizar las posteriores simulaciones y predecir como se comportaría una red mayor como Internet.

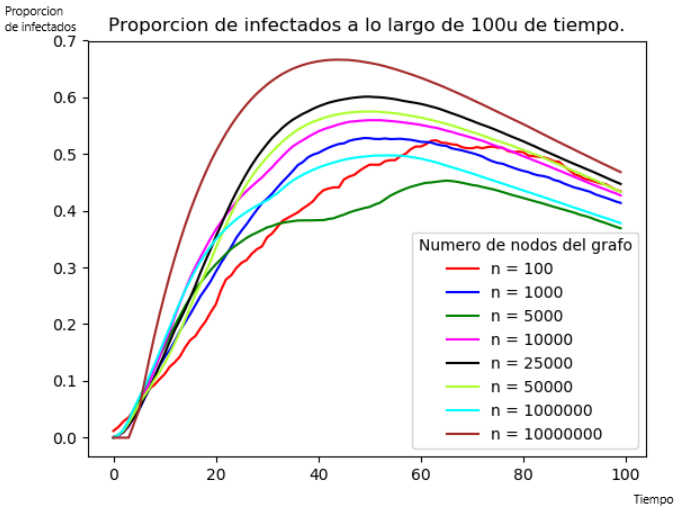


Figura 5.1: Gráfica correspondiente a la Simulación 1. Se puede ver la simulación en el Código B.1.1.

5.2. Relevancia del *paciente cero*

A lo largo de este documento, hemos introducido el concepto de *hub*, un nodo perteneciente a una red cuyo grado es mucho mayor que el grado medio. A la hora de analizar la propagación del *malware* nos preguntamos la influencia de empezar en un nodo de estas características frente a una infección aleatoria. Por razones obvias, estos nodos suelen tener un mayor grado de seguridad, por lo que surge la siguiente pregunta: ¿Es, desde el punto de vista del criminal, rentable tomar un *hub* como paciente cero?

Para responder esta pregunta nos hemos puesto en un caso extremo y hemos realizado dos simulaciones. La primera es una simulación donde la epidemia empieza en el nodo de **mayor grado** de la red y la segunda es una simulación aleatoria como las del apartado anterior. En la Tabla 5.2 podemos ver los parámetros de las simulaciones.

# nodos	F. de propagación	Modelo	Ejecs.	Tiempo	β	γ	Infectados	Susceptibles
10^5	simulacionHub	SIR	10	100	0.025	0.01	Hub	99999
10^5	simulacion	SIR	10	100	0.025	0.01	1	99999

Tabla 5.2: Tabla con los parámetros de la Simulación 2.

Al igual que en la simulación anterior hemos promediado diez ejecuciones en cada simulación, pues al ser la segunda aleatoria podría empezar por azar en un *hub*. Si observamos la Figura 5.2, podemos observar cómo empezar la infección en un *hub* influye en la velocidad de propagación pero no tan notablemente en la proporción de infectados. Sin embargo, para los criminales el tiempo es crítico y han de infectar a la mayor población posible antes de que se repare la vulnerabilidad.

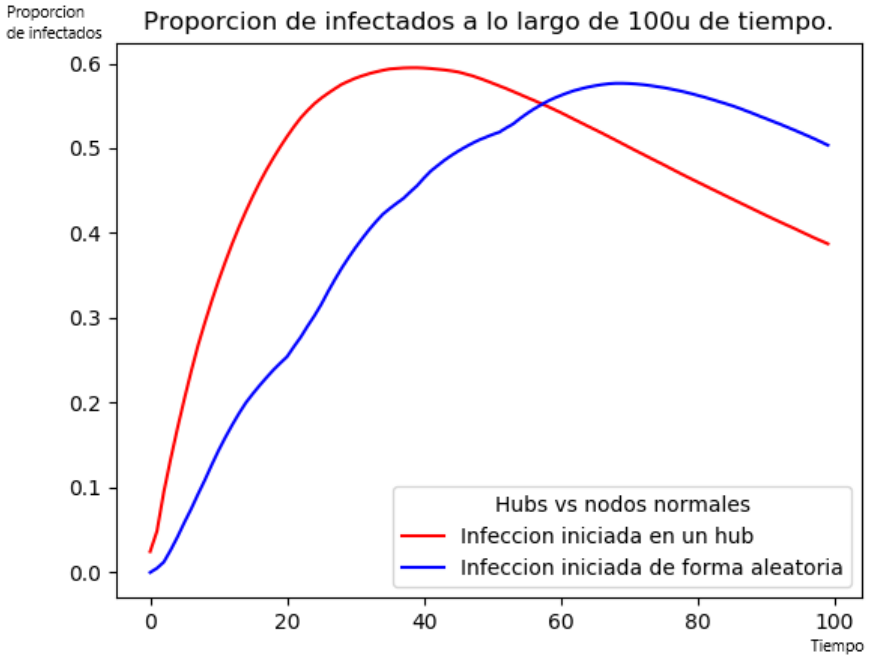


Figura 5.2: Gráfica correspondiente a la Simulación 2. Se puede ver la simulación en el Código B.1.2.

5.3. Infección dirigida a nodos de alto grado

En la simulación anterior, estudiamos como se comporta la epidemia si empieza en el nodo con mayor grado. Esta, supone que el software malicioso que se propaga tiene una inteligencia que le permite infectar con probabilidad uno a aquel nodo con el que entra en contacto en los cinco primeros turnos y tiene grado mayor que el grado medio del grafo.

Este escenario no es imposible de imaginar, pues dispositivos con funciones similares usan *software* similar, por lo que una vulnerabilidad en un *hub* podría afectar al resto. No obstante, el tiempo de respuesta es bajo, por eso suponemos que este *malware* lleva otro encapsulado que funciona de forma habitual.

Esto quiere decir, que el software malicioso de la primera simulación, realiza **los cinco primeros pasos** de forma dirigida a por los nodos de alto grado. Después de esto, el algoritmo de propagación es el mismo que en la infección aleatoria.

Como en el apartado anterior, realizaremos dos simulaciones. Una con el software malicioso inteligente, y el otro con la función simulación básica. Podemos ver los parámetros de las simulaciones en la Tabla 5.3.

# nodos	F. de propagación	Modelo	Ejecs.	Tiempo	β	γ	Infectados	Susceptibles
10^5	simulacionInteligente	XSIR	10	100	0.025	0.01	1	99999
10^5	simulacion	XSIR	10	100	0.025	0.01	1	99999

Tabla 5.3: Tabla con los parámetros de la Simulación 3.

En la Figura 5.3 podemos ver que esta supuesta inteligencia afectaría tanto al máximo nodo de infectados como a la velocidad de propagación. Junto a los resultados de la simulación anterior, podemos entonces afirmar que la seguridad en los nodos de mayor grado de una red debería ser proporcional a su grado para disminuir el impacto de la epidemia.

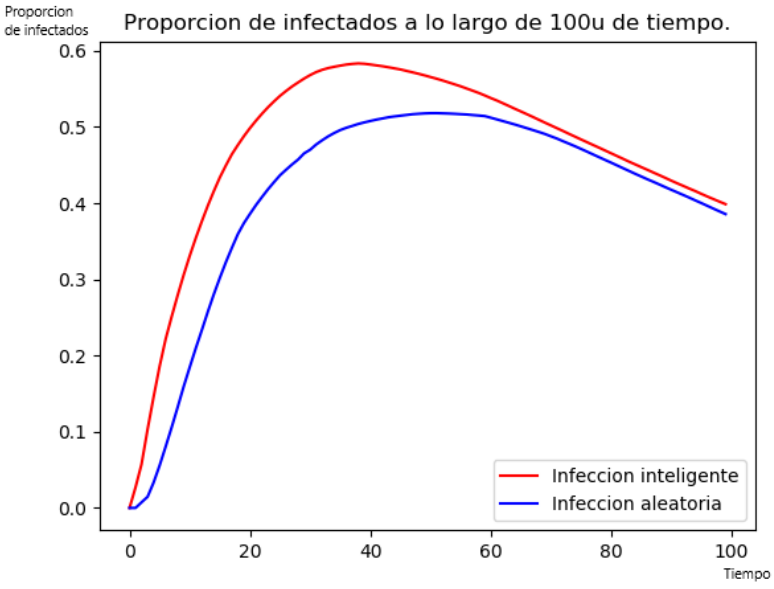


Figura 5.3: Gráfica correspondiente a la Simulación 3. Se puede ver la simulación en el Código B.1.3.

5.4. Visualización de una epidemia con parámetros inferidos de un caso real

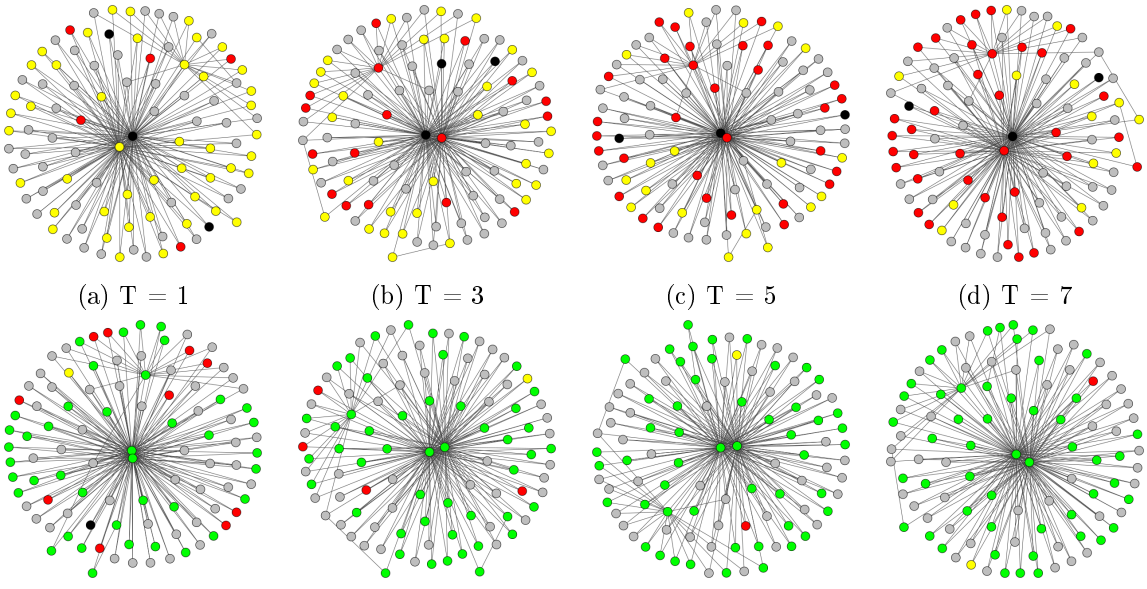
En la simulación siguiente, se intenta replicar lo sucedido el verano de 2017. Durante este verano, una epidemia de *malware* azotó distintos lugares del planeta, afectando desde particulares a grandes multinacionales. Las especificaciones del *malware*, se encuentran resumidas en la Sección 3.3.1 y la red es una *scale free* como en las demás simulaciones pero solo con 100 nodos.

Para la visualización de esta simulación, se ha decidido realizar una sola ejecución. Esta decisión se tomó para poder realizar impresiones del grafo cada dos unidades de tiempo.

Antes de imprimir o *fotografiar* el grafo en un momento concreto, se le aplica una capa de colores de forma que se puedan diferenciar los distintos compartimentos. Los colores corresponden a la siguiente leyenda:

- Gris = Nodos en el compartimento X, ajenos a la infección.
- Negro = Pacientes cero
- Verde = Nodos recuperados
- Amarillo = Nodos susceptibles
- Rojo = Nodos infectados

En la Figura 5.5, podemos observar como antes del lanzamiento del parche que repara la vulnerabilidad en $T = 8$, la mayoría de los nodos con posibilidad de infectarse se infectan. Esto concuerda con la realidad puesto que la epidemia de *WannaCry* actúo de forma virulenta en la primera semana desde el comienzo del estallido y después hubo una recuperación considerablemente rápida [68].



(a) T = 1 (b) T = 3 (c) T = 5 (d) T = 7

(e) T = 9 (f) T = 11 (g) T = 13 (h) T = 15

Figura 5.4: Estado del grafo en distintos momentos de la epidemia. Se puede ver la generación en el Código B.1.5.

5.5. Conjunto inicial de infectados junto vs disperso

En la penúltima simulación de este documento, se busca ver la relevancia de que el grupo de infectados inicial este junto o disperso. Aparentemente, parece que si los infectados iniciales están dispersos, podrán llegar a distintos nodos cada uno y por tanto, infectar a mayor parte de la población. Bajo esta premisa realizamos la simulación.

Los parámetros de la simulación se pueden observar en la Tabla 5.4.

# nodos	F. de propagación	Modelo	Ejecs.	Tiempo	β	γ	Infectados	Susceptibles
10^5	simulacionGrupoInicial	SIR	10	100	0.025	0.01	3	99997
10^5	simulacion	SIR	10	100	0.025	0.01	3	99997

Tabla 5.4: Tabla con los parámetros de la Simulación 5.

En vez de lo esperado, la Figura 5.5 nos muestra un comportamiento antiintuitivo. El grupo que comienza unido se propaga de forma más veloz y alcanza una mayor proporción de infectados. Esto se debe al algoritmo mediante el que se escogen los infectados, que selecciona un nodo aleatorio y otros dos de sus vecinos como primeros infectados. Al fijar un nodo y seleccionar dos de sus vecinos, hay una alta probabilidad de que uno de los dos vecinos tenga un alto grado de conexión y pertenezca a otra comunidad dando lugar a los resultados obtenidos.

Otra anomalía de la gráfica es que ambas simulaciones no comienzan en el mismo punto. Esto se debe a que el primer dato que se refleja es el obtenido trás la primera iteración de la simulación. Esto quiere decir que el primer punto de la gráfica no es el estado inicial, sino el estado resultante de aplicarle el primer paso de la simulación.

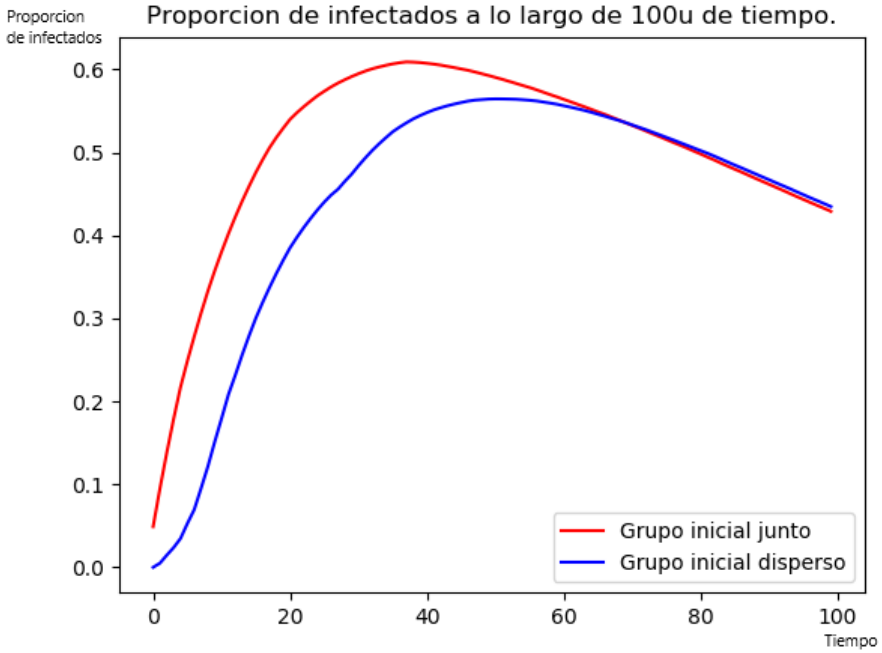


Figura 5.5: Gráfica correspondiente a la Simulación 5. Se puede ver la simulación en el Código B.1.6.

5.6. Tasa de infección β fijo, tasa de recuperación γ variable

Como mencionamos en la Sección 2.3, el modelo SIR de epidemiología depende de dos parámetros, β y γ. En esta simulación vamos a ver como influye γ en la propagación del *malware*.

Recordamos que $\beta = 0{,}025$ era la tasa de infección y γ la tasa de recuperación. Para este apartado hemos realizado seis simulaciones con los mismos parámetros salvo γ que toma valores entre 0 y β.

Como vemos en la Figura 5.6 las simulaciones responden de la manera esperada, es decir, a mayor tasa de recuperación menor es la velocidad de propagación así como la mayor proporción de infectados.

Podemos observar en la gráfica que existen algunas irregularidades como para $\gamma = 0{,}015$ que es inferior a los valores 0.02 y 0.025 cuando debería ser así. Esto es debido a las limitaciones de *hardware* y se solucionaría si en vez de usar el promedio de 10 ejecuciones, usasemos un promedio de un mayor número de ejecuciones.

Otro fenómeno curioso y que merece mencionarse, es la apreciable diferencia entre las tres gráficas de valor superior y las tres gráficas de valor menor. Este cambio se debe a que el punto de inflexión de la ecuación de infección se halla entre los dos valores medios [69].

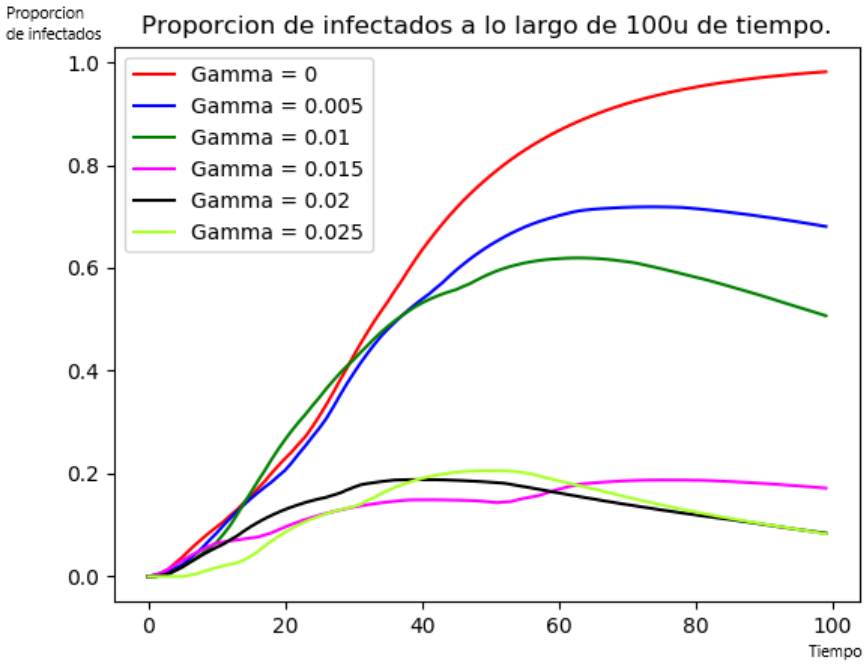


Figura 5.6: Gráfica correspondiente a la Simulación 6. Se puede ver la simulación en el Código B.1.7.

6

Conclusiones y trabajo futuro

6.1. Conclusiones

Este trabajo ha sido un acercamiento al mundo de la epidemiología de *malware*. En él se han estudiado las distintas disciplinas que intervienen en este proceso. Desde el estudio del *malware*, su clasificación según sus características y el impacto del mismo hasta la estructura topológica que posee Internet y su influencia en el proceso. También se han estudiado los distintos modelos utilizados, tanto en la epidemiología biológica como en la de software malicioso, centrandonos en los modelos compartimentales, más comunes a la hora de representar epidemias.

Una vez adquiridos los conocimientos necesarios para entender cómo se propaga la información, y en particular el *malware*, en una red de tipo *scale free*; se ha diseñado una herramienta utilizando diversas tecnologías existentes de programación y especializadas en el manejo y análisis de redes.

Trás desarrollar la herramienta y contar con todo lo necesario para realizar la simulación, se pensaron una serie de preguntas sobre cómo se comporta la epidemia y la red ante distintas situaciones.

Estas simulaciones nos permitieron obtener respuestas y un mayor conocimiento sobre el tema, y señalaron distintos hechos a tener en cuenta a la hora de predecir cómo se propagará un determinado *malware* en Internet.

Esto reafirma el pensamiento del autor de **la necesidad de continuar con esta línea de investigación a fin de realizar estrategias preventivas adecuadas y tomar las contramedidas correctas en caso de epidemia**.

6.2. Trabajo futuro

La línea de investigación de este trabajo, es una línea en continuo desarrollo y muchos de los retos a resolver están ya planteados o actualmente siendo investigados. Algunos de los temas que se deberían estudiar a continuación de este trabajo, son los siguientes:

- Conseguir un mapa actualizado de Internet.

- Parametrizar distintos *malwares* reales de forma adecuada.
- Implementar una interfaz gráfica para la herramienta desarrollada además de permitir la parametrización desde la misma.
- Habilitar la posibilidad de realizar la simulación de propagación sobre una red en crecimiento de forma que sea más similar a Internet.

Bibliografía

[1] MAJ William R. Detlefsen. Cyber attacks, Attribution, and Deterrence: Three Case Studies. Technical report, Computer Security Institute, May 2011.

[2] FirefoxEmoji. http://mozilla.github.io/fxemoji/dist/FirefoxEmoji/index.html. Accedido en 2018-06-06.

[3] John Aycock. *Computer viruses and malware.* Number 22 in Advances in information security. Springer, New York, NY, 2006.

[4] William Stallings and Lawrie Brown. *Computer security principles and practice.* Pearson, Boston, 2015.

[5] Archit Gupta, Pavan Kuppili, Aditya Akella, and Paul Barford. An empirical study of malware evolution. pages 1–10. IEEE, January 2009.

[6] Gunter Ollmann. The evolution of commercial malware development kits and colour-by-numbers custom malware. *Computer Fraud & Security*, 2008(9):4–7, September 2008.

[7] *Mathematical Models for Malware Propagation.*

[8] Luc Tidy, Khurram Shahzad, Muhammad Aminu Ahmad, and Steve Woodhead. *An assessment of the contemporary threat posed by network worm malware.* IARIA, Nice, France, October 2014.

[9] python-igraph. http://igraph.org/python/. Accedido en 2018-06-17.

[10] Edoardo M. Airoldi, Tamás Nepusz, and Gábor Csárdi. *Statistical Network Analysis with igraph.* Springer.

[11] *NotPetya Attack Costs Big Companies Millions SecurityWeek.Com.* Accedido en 2018-05-26.

[12] ENISA. ENISA Threat Landscape Report 2017. Technical report, January 2018.

[13] MalwarebytesLABS. Cybercrime tactics and techniques: 2017 state of malware. Technical report.

[14] Obama Ordered Wave of Cyberattacks Against Iran - The New York Times. https://www.nytimes.com/2012/06/01/world/middleeast/obama-ordered-wave-of-cyberattacks-against-iran.html. Accedido en 2018-06-03.

[15] Top Georgian Official: Moscow Cyber Attacked Us We Just Can't Prove It WIRED. https://www.wired.com/2009/03/georgia-blames/. Accedido en 2018-06-03.

[16] MAJ William R. Detlefsen. Cyber attacks, Attribution, and Deterrence: Three Case Studies. Technical report, U.S. Army Command and General Staff College, May 2015.

[17] Murugiah Souppaya and Karen Scarfone. Guide to Malware Incident Prevention and Handling for Desktops and Laptops. Technical Report NIST SP 800-83r1, National Institute of Standards and Technology, July 2013.

[18] Spectre (vulnerabilidad) - Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Spectre_(vulnerabilidad). Accedido en 2018-05-26.

[19] The war against mobile 'adware' isn't over yet, warns Lookout \textbackslashtextbar Technology \textbackslashtextbar The Guardian. https://www.theguardian.com/technology/2014/feb/21/mobile-adware-chargeware-lookout-2013. Accedido en 2018-06-06.

[20] Rory Cellan-Jones. Malware mastermind suspect arrested. Russian police reportedly arrest a man on suspicion of masterminding two infamous hacking tools., October 2013. Accedido en 2018-06-06.

[21] How does auto-rooting malware LevelDropper gain device root access? https://searchsecurity.techtarget.com/answer/How-does-auto-rooting-malware-LevelDropper-gain-device-root-access. Accedido en 2018-06-06.

[22] Hackers Hid Backdoor In CCleaner Security App With 2 Billion Downloads 2.3 Million Infected. https://www.forbes.com/sites/thomasbrewster/2017/09/18/ccleaner-cybersecurity-app-infected-with-backdoor/#76a1d72a316a. Accedido en 2018-06-06.

[23] CertUtil.exe Could Allow Attackers To Download Malware While Bypassing AV. https://www.bleepingcomputer.com/news/security/certutilexe-could-allow-attackers-to-download-malware-while-bypassing-av/. Accedido en 2018-06-06.

[24] Trojan.Downloader. https://blog.malwarebytes.com/detections/trojan-downloader/. Accedido en 2018-06-06.

[25] Proyecto ESCEMMat UCM. http://www.mat.ucm.es/imgomezc. Accedido en 2018-05-21.

[26] Angel Martín del Rey. Mathematical modeling of the propagation of malware: a review: Mathematical modeling of the propagation of malware: a review. *Security and Communication Networks*, 8(15):2561–2579, October 2015.

[27] CAIDA: Center for Applied Internet Data Analysis. CAIDA Interactive - Overview of CAIDA Interactive Services. http://www.caida.org/interactive/index.xml. Accedido en 2018-05-21.

[28] Socilab - LinkedIn Social Network Visualization, Analysis, and Education. http://socilab.com/#home. Accedido en 2018-05-21.

[29] The Opte Project. http://www.opte.org/. Accedido en 2018-05-21.

[30] H.T. Banks, Jared Catenacci, and Shuhua Hu. Stochastic vs. Deterministic Models for Systems with Delays. *IFAC Proceedings Volumes*, 46(26):61–66, 2013.

[31] Howard M. Taylor and Samuel Karlin. *An introduction to stochastic modeling*. Academic Press, San Diego, 3rd ed edition, 1998.

[32] Ángel Martín Del Rey, F. K. Batista, and A. Queiruga Dios. Malware propagation in Wireless Sensor Networks: global models vs Individual-based models. *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal*, 6(3):5, September 2017.

[33] Daniel Bernoulli and Sally Blower. An attempt at a new analysis of the mortality caused by smallpox and of the advantages of inoculation to prevent it. *Reviews in Medical Virology*, 14(5):275–288, September 2004.

[34] K. Ramachandran and B. Sikdar. Modeling malware propagation in Gnutella type peer-to-peer networks. page 8 pp. IEEE, 2006.

[35] Jennifer T. Jackson and Sadie Creese. Virus Propagation in Heterogeneous Bluetooth Networks with Human Behaviors. *IEEE Transactions on Dependable and Secure Computing*, 9(6):930–943, November 2012.

[36] Wanping Liu, Chao Liu, Zheng Yang, Xiaoyang Liu, Yihao Zhang, and Zuxue Wei. Modeling the propagation of mobile malware on complex networks. *Communications in Nonlinear Science and Numerical Simulation*, 37:249–264, August 2016.

[37] Frank L. Smith. Malware and Disease: Lessons from Cyber Intelligence for Public Health Surveillance. *Health Security*, 14(5):305–314, October 2016.

[38] SI and SIS models Generic Model documentation. Accedido en 2018-05-25.

[39] SIR and SIRS models Generic Model documentation. https://institutefordiseasemodeling.github.io/Documentation/general/model-sir.html. Accedido en 2018-05-25.

[40] SEIR and SEIRS models Generic Model documentation. https://institutefordiseasemodeling.github.io/Documentation/general/model-seir.html. Accedido en 2018-05-25.

[41] Pablo Fernández Gallardo José Luis Fernández Pérez. Capítulo 8. Grafos. In *El discreto encanto de la matemática(Sin publicar)*. https://www.uam.es/personal_pdi/ciencias/gallardo/capitulo8a.pdf.

[42] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, August 2007.

[43] A.-L. Barabasi. Scale-Free Networks: A Decade and Beyond. *Science*, 325(5939):412–413, July 2009.

[44] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393:440, June 1998.

[45] Coeficiente de agrupamiento, November 2017. Accedido en 2018-06-06.

[46] F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009.

[47] James J. Collins and Carson C. Chow. It's a small world. *Nature*, 393(6684):409–410, June 1998.

[48] Duncan J. Watts. *Small worlds: the dynamics of networks between order and randomness*. Princeton studies in complexity. Princeton Univ. Press, Princeton, NJ, 8. print., 1. paperback print edition, 2004.

[49] P ErdÃűs. P. Erdös and A. Rényi, Publ. Math.(Debrecen) 6, 290 (1959). *Publ. Math.(Debrecen)*, 6:290, 1959.

[50] Fred Schepisi. *Seis grados de separaciÃşn*. June 1995.

[51] Vito Latora and Massimo Marchiori. Efficient Behavior of Small-World Networks. *Physical Review Letters*, 87(19), October 2001.

[52] Li Fu, Wenjie Huang, Sheng Xiao, Yuan Li, and Shifan Guo. Vulnerability Assessment for Power Grid Based on Small-world Topological Model. pages 1–4. IEEE, 2010.

[53] Albert-LászlÃş Barabási. *Linked: the new science of networks*. Perseus Pub, Cambridge, Mass, 2002.

[54] RÃľka Albert and Albert-LászlÃş Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, January 2002.

[55] *The Oracle of Bacon*.

[56] RÃľka Albert and Albert-LászlÃş Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, January 2002.

[57] graph-tool: Efficent network analysis with python. https://graph-tool.skewed.de/. Accedido en 2018-06-18.

[58] NetworkX. https://networkx.github.io/. Accedido en 2018-06-18.

[59] Yi Wang, Zhen Jin, Zimo Yang, Zi-Ke Zhang, Tao Zhou, and Gui-Quan Sun. Global analysis of an SIS model with an infective vector on complex networks. *Nonlinear Analysis: Real World Applications*, 13(2):543–557, April 2012.

[60] Ranjit Kumar Upadhyay, Sangeeta Kumari, and A. K. Misra. Modeling the virus dynamics in computer network with SVEIR model and nonlinear incident rate. *Journal of Applied Mathematics and Computing*, 54(1-2):485–509, June 2017.

[61] Hong Guo, Hsing Kenneth Cheng, and Ken Kelley. Impact of Network Structure on Malware Propagation: A Growth Curve Perspective. *Journal of Management Information Systems*, 33(1):296–325, January 2016.

[62] Sneha S and Malathi L. A Survey on Malware Propagation Analysis and Prevention Model. *International Journal of Advancements in Technology*, 06(02), 2015.

[63] Michael Sanford. Computer viruses and malware by John Aycock. *ACM SIGACT News*, 41(1):44, March 2010.

[64] *Graph theory*. Springer Berlin Heidelberg, New York, NY, 2017.

[65] RÃľka Albert, Hawoong Jeong, and Albert-LászlÃş Barabási. Diameter of the World-Wide Web: Internet. *Nature*, 401(6749):130–131, September 1999.

[66] WannaCry Ransomware Statistics: The Numbers Behind the Outbreak. https://blog.barkly.com/wannacry-ransomware-statistics-2017. Accedido en 2018-06-18.

[67] Lian-Ming Zhang, Xiao-Heng Deng, Jian-Ping Yu, and Xiang-Sheng Wu. Degree and connectivity of the Internet's scale-free topology. *Chinese Physics B*, 20(4):048902, April 2011.

[68] 'WannaCry File Restore', la herramienta de Telefónica para recuperar archivos afectados por el ransomware. https://www.xataka.com/aplicaciones/wannacry-file-restore-la-herramienta-de-telefonica-para-recuperar-archivos-afectados-por-el-ransomware. Accedido en 2018-06-20.

[69] Aiguo Duan, Jianguo Zhang, Xiongqing Zhang, and Caiyun He. Relationship between Modelling Accuracy and Inflection Point Attributes of Several Equations while Modelling Stand Diameter Distributions. *PLOS ONE*, 10(5):e0126831, May 2015.

A

Clasificación de *Malware*

En la Tabla A.1 podemos ver una clasificación de los distintos tipos de *malware* de acuerdo a su capacidad de auto-replicación, el crecimiento de la población de infectados y la capacidad del software malicioso de ejecutarse de forma independiente.

Nombre	Auto-replicación	Crec. de Población	Parásito	Descripción
Adware [19]	No	0	Sí	Software que crea publicidad indeseada. Puede redirigir el navegador a sitios comerciales o abrir *pop-ups*.
Kit de ataque [20]	No	0	No	Kit de herramientas para la generación de un amplio espectro de malware.
Auto-rooter [21]	No	0	No	Kit de herramientas para forzar la entrada en otro sistema.
Backdoor [22]	No	0	Sí	Un mecanismo para pasar controles de seguridad y acceder a otro programa o sistema.
Downloaders [23] [24]	No	0	Sí	*Malware parásito* que suele servir para descargar otro paquete de *malware*.
Drive-by-download	No	0	Sí	*Malware* encontrado normalmente en una web comprometida que permite atacar a los visitantes de la página.
Exploits	A veces	Depende	A veces	*Malware* que ataca aprovechándose de ciertas vulnerabilidades conocidas.
Flooders	A veces	Depende	A veces	*Malware* que se usa para crear grandes volúmenes de datos y realizar ataques DoS en redes.
Keyloggers	No	0	Solo en la instalación	*Malware* que captura la entrada de teclado(u otros dispositivos) del sistema infectado.
Logic bomb	No	0	A veces	*Malware* instalado por un intruso que permanece latente hasta cumplir una condición que lo dispare.
Mobile code	A veces	Depende	A veces	*Malware* preparado en un entorno aislado capaz de ejecutarse en distintos sistemas.
Rootkit	No	0	No	Conjunto de herramientas que permiten al intruso, una vez dentro del sistema, ganar acceso de *root*.
Spammer programs	No	0	No	*Malware* encargado de enviar grandes cantidades de *spam*.
Spyware	No	0	No	*Malware* que recolecta información sensible del sistema infectado.
Trojan horse [24]	No	0	Sí	*Malware* que se esconde en un programa de apariencia útil y permite sortear controles de seguridad del ordenador infectado.
Worm	Sí	Positivo	No	*Malware* que suele ejecutarse por sí mismo y replicarse. Normalmente utilizando un exploit.
Zombie, bot	A veces	Depende	No	*Malware* que permanece en estado latente hasta ser activado por el intruso, normalmente junto a otro gran número de sistemas infectados, para realizar un ataque.

Tabla A.1: Clasificación de *malware*. Tabla adaptada de: [3][4]

B

Códigos del entorno de simulación

B.1. Simulaciones

B.1.1. Simulacion1.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
from FuncionesNucleo import FuncionesNucleo
from ModeloSIR import ModeloSIR
import matplotlib.pyplot as plt
from igraph import *


class Simulacion1(object):

    def main(self):
        """En esta primera simulacion vamos a ver como influye el tamanyo
        de la poblacion en el porcentaje de poblacion en cada compartimento.
        Para ello realizaremos la simulacion sobre un grafo en un periodo de
        100 unidades de tiempo para tamanyos desde 100 a 1000000."""

        tam = [100, 1000, 5000, 10000, 25000, 50000, 100000, 1000000]
        tiempo = 100
        beta = 0.025
        gamma = 0.01
        resultados = []

        for s in tam:
            print "Simulacion 1 Tamanyo: %d" %s
            if s != 1000000:
                aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR,
                                                         "Scale_Free",
                                                         tam_grafo=s,
```

```
                                                    time=tiempo,
                                                    beta=beta, gamma=gamma)

            ## CdE
            media = [mean(items) for items in izip(*aux2)]
            while abs(mean(media)) - 0.05 < 0:
                aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR,
                                                    "Scale_Free",
                                                    tam_grafo=s,
                                                    time=tiempo,
                                                    beta=beta,
                                                    gamma=gamma)
                media = [mean(items) for items in izip(*aux2)]
            # Anyadimos el resultado
            resultados.append(aux)

            # La simulacion con un millon de nodos solo la haremos una vez
            # por limitaciones del HW.
        else:
            aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR,
                                                    "Scale_Free",
                                                    tam_grafo=s, runs=1,
                                                    time=tiempo, beta=beta,
                                                    gamma=gamma)

            ## CdE
            media = [mean(items) for items in izip(*aux2)]
            while abs(mean(media)) - 0.05 < 0:
                aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR,
                                                    "Scale_Free",
                                                    tam_grafo=s,
                                                    time=tiempo,
                                                    beta=beta,
                                                    gamma=gamma)
                media = [mean(items) for items in izip(*aux2)]
            # Anyadimos el resultado
            resultados.append(aux)

    tiempo = range(tiempo)
    averaged = [mean(items) for items in izip(*resultados[0])]
    averaged2 = [mean(items) for items in izip(*resultados[1])]
    averaged3 = [mean(items) for items in izip(*resultados[2])]
    averaged4 = [mean(items) for items in izip(*resultados[3])]
    averaged5 = [mean(items) for items in izip(*resultados[4])]
    averaged6 = [mean(items) for items in izip(*resultados[5])]
    averaged7 = [mean(items) for items in izip(*resultados[6])]
    averaged8 = [mean(items) for items in izip(*resultados[7])]

    plt.plot(tiempo, averaged, 'r', label='n = 100')
    plt.plot(tiempo, averaged2, 'b', label='n = 1000')
    plt.plot(tiempo, averaged3, 'g', label='n = 5000')
```

```
        plt.plot(tiempo, averaged4, 'fuchsia', label='n = 10000')
        plt.plot(tiempo, averaged5, 'black', label='n = 25000')
        plt.plot(tiempo, averaged6, 'greenyellow', label='n = 50000')
        plt.plot(tiempo, averaged7, 'cyan', label='n = 1000000')
        plt.plot(tiempo, averaged8, 'brown', label='n = 10000000')
        plt.legend(title="Numero de nodos del grafo")
        plt.title("Proporcion de infectados a lo largo de 100u de tiempo.")
        #plt.show()
        plt.savefig('Figuras/simulacion1.png')


ejec = Simulacion1()
ejec.main()
```

B.1.2. Simulacion2.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
from FuncionesNucleo import FuncionesNucleo
from ModeloSIR import ModeloSIR
import matplotlib.pyplot as plt
from igraph import *


class Simulacion2(object):

    def main(self):
        """ En esta segunda simulación vamos a comprobar como afecta
         en una epidemia si el virus empieza en un hub o no. Para ello
         utilizaremos los mismos parámetros que en el apartado anterior
         a la hora de realizar la simulacion, salvo a la hora de escoger el
         primer nodo infectado."""

        tam = 100000
        tiempo = 100
        beta = 0.025
        gamma = 0.01
        resultados = []

        print "Simulacion 1"
        aux, aux2 = FuncionesNucleo().simulacionHub(ModeloSIR,
                                                    "Scale_Free",
                                                    tam_grafo=tam,
                                                    time=tiempo, beta=beta,
                                                    gamma=gamma)
        ## CdE
        media = [mean(items) for items in izip(*aux2)]
        while abs(mean(media)) - 0.05 < 0:
            aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR, "Scale_Free",
                                                     tam_grafo=tam, time=tiempo,
                                                     beta=beta, gamma=gamma)
            media = [mean(items) for items in izip(*aux2)]
```

```
        resultados.append(aux)

        print "Simulacion 2"
        aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR, "Scale_Free",
                                                 tam_grafo=tam,
                                                 time=tiempo, beta=beta,
                                                 gamma=gamma)
        ## CdE
        media = [mean(items) for items in izip(*aux2)]
        while abs(mean(media)) - 0.05 < 0:
            aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR, "Scale_Free",
                                                     tam_grafo=tam, time=tiempo,
                                                     beta=beta, gamma=gamma)
            media = [mean(items) for items in izip(*aux2)]

        resultados.append(aux)

        tiempo = range(tiempo)
        averaged = [mean(items) for items in izip(*resultados[0])]
        averaged2 = [mean(items) for items in izip(*resultados[1])]

        plt.plot(tiempo, averaged, 'r',
                 label='Infeccion iniciada en un hub')
        plt.plot(tiempo, averaged2, 'b',
                 label='Infeccion iniciada de forma aleatoria')
        plt.legend(title="Hubs vs nodos normales")
        plt.title("Proporcion de infectados a lo largo de 100u de tiempo.")
        # Plot
        plt.show()
        #plt.savefig('Figuras/simulacion2.png')


ejec = Simulacion2()
ejec.main()
```

B.1.3. Simulacion3.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
from FuncionesNucleo import *
from ModeloXSIR import ModeloXSIR
import matplotlib.pyplot as plt
from igraph import *


class Simulacion3(object):

    def main(self):
        """ En esta simulacion procederemos a ver cuanto mejora
        un gusano si le metemos intelegencia. En nuestra simulación,
        usaremos un nodo de tam 10^5, en el que supondremos como tres
```

```
    cuartas partes de la poblacion es susceptible, y empieza con tres
    infectados. Estos tres infectados tendrán probabilidad 1 de infectar
    a nodos con un grado alto (perc. 90) en los 5 primeros contactos, y
    0 en caso contrario. Después la epidemia se ejecuta con los valores
    de transición de las simulaciones anteriores. Por otra parte, se
    simulará la misma prueba sin contar esta inteligencia."""

    tam = 10 ** 5
    tiempo = 100
    beta = 0.025
    gamma = 0.01
    resultados = []

    """ Simulacion 1 """
    print "Simulacion 1"
    aux, aux2 = FuncionesNucleo().simulacionInteligente(ModeloXSIR,
                                                        "Scale_Free",
                                                        tam_grafo=tam,
                                                        susceptibles=tam,
                                                        n_infectados=5,
                                                        time=tiempo,
                                                        beta=beta,
                                                        gamma=gamma)
    ## CdE
    media = [mean(items) for items in izip(*aux2)]
    i = 2
    while abs(mean(media)) - 0.005 < 0:
        print "Intento %d" % i
        print "Media: %.3f" % abs(mean(media))
        i += 1
        aux, aux2 = FuncionesNucleo().simulacionInteligente(ModeloXSIR,
                                                            "Scale_Free",
                                                            tam_grafo=tam,
                                                            susceptibles=tam,
                                                            n_infectados=5,
                                                            time=tiempo,
                                                            beta=beta,
                                                            gamma=gamma)
        media = [mean(items) for items in izip(*aux2)]

    resultados.append(aux)

    t = range(tiempo)
    averaged = [mean(items) for items in izip(*resultados[0])]
    plt.plot(t, averaged, 'r', label='Infeccion inteligente')

    """ Simulacion 2 """
    print "Simulacion 2"
    aux, aux2 = FuncionesNucleo().simulacion(ModeloXSIR,
                                             "Scale_Free", tam_grafo=tam,
                                             susceptibles=tam,
```

```
                                                    n_infectados=5,
                                                    time=tiempo, beta=beta,
                                                    gamma=gamma)
        ## CdE
        media = [mean(items) for items in izip(*aux2)]
        i = 2
        while abs(mean(media)) - 0.005 < 0:
            print "Intento %d" % i
            print "Media: %.3f" % abs(mean(media))
            i += 1
            aux, aux2 = FuncionesNucleo().simulacion(ModeloXSIR,
                                                    "Scale_Free",
                                                    tam_grafo=tam,
                                                    susceptibles=tam,
                                                    n_infectados=5,
                                                    time=tiempo, beta=beta,
                                                    gamma=gamma)
            media = [mean(items) for items in izip(*aux2)]

        resultados.append(aux)

        t = range(tiempo)
        averaged2 = [mean(items) for items in izip(*resultados[1])]
        plt.plot(t, averaged2, 'b', label='Infeccion aleatoria')

        """ Generar grafica """
        plt.title("Proporcion de infectados a lo largo de %du de tiempo."
                  % tiempo)
        plt.legend()
        plt.show()
        # plt.savefig('Figuras/simulacion3.png')


ejec = Simulacion3()
ejec.main()
```

B.1.4. Simulacion4.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
import math
from FuncionesNucleo import *
from ModeloSIS import ModeloSIS
import matplotlib.pyplot as plt

""" En esta simulacion procederemos a ver cuanto mejora
un gusano si le metemos intelegencia. En nuestra simulación,
usaremos un nodo de tam 10^5, en el que supondremos como tres
cuartas partes de la poblacion es susceptible, y empieza con tres
infectados.
Estos tres infectados tendrán probabilidad 1 de infectar a nodos con mayor
grado que la media en los 5 primeros contactos, y 0 en caso contrario.
```

```
 Después
la epidemia se ejecuta con los valores de transición de las simulaciones
anteriores. Por otra parte, se simulará la misma prueba sin contar esta inteligen


tam = 10 ** 5
resultados = []

""" Simulacion 1 """
resultados.append(simulacionInteligente2(ModeloSIS, "Scale_Free", tam_grafo=tam,

tiempo = range(100)
averaged = [mean(items) for items in izip(*resultados[0])]
plt.plot(tiempo, averaged, 'r', label='Infeccion inteligente')



""" Simulacion 2 """
resultados.append(simulacion(ModeloSIS, "Scale_Free", tam_grafo=tam, susceptibles

tiempo = range(100)
averaged2 = [mean(items) for items in izip(*resultados[1])]
plt.plot(tiempo, averaged2, 'b', label='Infeccion aleatoria')

""" Generar grafica """
plt.title("Proporcion de infectados a lo largo de 100u de tiempo.")
plt.legend()
plt.savefig('simulacion4.png')
```

B.1.5. Simulacion5.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
import math
from FuncionesNucleo import FuncionesNucleo
from ModeloXSIR import ModeloXSIR
import matplotlib.pyplot as plt
from igraph import *


class Simulacion5(object):

    def main(self):
        """ En esta simulación colorearemos los compartimentos distintos
        para ver de una forma visual la propagación del malware, en un caso
        con parametros obtenidos del caso 'Wannacry'."""
        tam = 10 ** 2
        tiempo = 100
        beta = 1
        gamma = 0.07
        resultados = []
```

```
        """ Simulacion 1 """
        resultados.append(FuncionesNucleo().simulacionColores(ModeloXSIR,
                        "Scale_Free", tam_grafo=tam, susceptibles=tam*0.5,
                        n_infectados=3, time=tiempo, beta=beta, gamma=gamma,
                        runs=1))

ejec = Simulacion5()
ejec.main()
```

B.1.6. Simulacion6.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
import math
from FuncionesNucleo import FuncionesNucleo
from ModeloSIR import ModeloSIR
import matplotlib.pyplot as plt
from igraph import *


class Simulacion6(object):

    def main(self):
        """ En esta simulación comprobaremos como afecta que los primeros
         infectados esten juntos o no. """
        tam = 10 ** 5
        tiempo = 100
        beta = 0.025
        gamma = 0.01
        runs = 10
        resultados = []

        print "Simulacion 1"
        aux, aux2 = FuncionesNucleo().simulacionGrupoInicial(ModeloSIR,
                                                             "Scale_Free",
                                                             tam_grafo=tam,
                                                             n_infectados=3,
                                                             time=tiempo,
                                                             beta=beta,
                                                             gamma=gamma,
                                                             runs=runs)
        ## CdE
        # media = [mean(items) for items in izip(*aux2)]
        # while abs(mean(media)) - 0.05 < 0:
        #     aux, aux2 = FuncionesNucleo().simulacionGrupoInicial(ModeloXSIR,
        #                 "Scale_Free", tam_grafo=tam, susceptibles=tam*0.8,
        #                 n_infectados=3, time=tiempo, beta=beta, gamma=gamma,
        #                 runs=runs)
        #     media = [mean(items) for items in izip(*aux2)]

        resultados.append(aux)
```

```
        print "Simulacion 2"
        aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR,
                                                 "Scale_Free", tam_grafo=tam,
                                                 n_infectados=3, time=tiempo,
                                                 beta=beta, gamma=gamma,
                                                 runs=runs)
        ## CdE
        # media = [mean(items) for items in izip(*aux2)]
        # while abs(mean(media)) - 0.05 < 0:
        #     aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR,
        #                 "Scale_Free", tam_grafo=tam, susceptibles=tam*0.8,
        #                 n_infectados=3, time=tiempo, beta=beta, gamma=gamma,
        #                 runs=runs)
        #     media = [mean(items) for items in izip(*aux2)]

        resultados.append(aux)

        tiempo = range(tiempo)
        averaged = [mean(items) for items in izip(*resultados[0])]
        averaged2 = [mean(items) for items in izip(*resultados[1])]

        plt.plot(tiempo, averaged, 'r',
                 label='Grupo inicial junto')
        plt.plot(tiempo, averaged2, 'b',
                 label='Grupo inicial disperso')
        plt.title("Proporcion de infectados a lo largo de 100u de tiempo.")
        plt.legend()
        # Plot
        plt.show()
        # plt.savefig('Figuras/simulacion2.png')


ejec = Simulacion6()
ejec.main()
```

B.1.7. Simulacion7.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
from FuncionesNucleo import FuncionesNucleo
from ModeloSIR import ModeloSIR
import matplotlib.pyplot as plt
from igraph import *


class Simulacion7(object):

    def main(self):
        """En esta simulacion veremos la influencia de gamma en la
        propagación del malware. Fijaremos beta y realizaremos la
        simulacion para seis gammas distintas. """
```

```
        tam = 100000
        tiempo = 100
        beta = 0.025
        gamma = [0, 0.005, 0.01, 0.015, 0.02, 0.025]
        resultados = []

        i = 0
        for g in gamma:
            print "Simulacion %.3f" %g
            aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR, "Scale_Free",
                                                     tam_grafo=tam, time=tiempo,
                                                     beta=beta, gamma=g)


                ## CdE
            media = [mean(items) for items in izip(*aux2)]

            i = 0
            # while abs(mean(media)) - 0.05 < 0:
            #     print "Ronda %d" %i
            #     aux, aux2 = FuncionesNucleo().simulacion(ModeloSIR, "Scale_Free
            #                                              tam_grafo=tam, time=tiempo,
            #                                              beta=beta, gamma=g)
            #     media = [mean(items) for items in izip(*aux2)]
            #     i += 1
            # # Anyadimos el resultado
            resultados.append(aux)

        tiempo = range(tiempo)
        averaged = [mean(items) for items in izip(*resultados[0])]
        averaged2 = [mean(items) for items in izip(*resultados[1])]
        averaged3 = [mean(items) for items in izip(*resultados[2])]
        averaged4 = [mean(items) for items in izip(*resultados[3])]
        averaged5 = [mean(items) for items in izip(*resultados[4])]
        averaged6 = [mean(items) for items in izip(*resultados[5])]

        plt.plot(tiempo, averaged, 'r', label='Gamma = 0')
        plt.plot(tiempo, averaged2, 'b', label='Gamma = 0.005')
        plt.plot(tiempo, averaged3, 'g', label='Gamma = 0.01')
        plt.plot(tiempo, averaged4, 'fuchsia', label='Gamma = 0.015')
        plt.plot(tiempo, averaged5, 'black', label='Gamma = 0.02')
        plt.plot(tiempo, averaged6, 'greenyellow', label='Gamma = 0.025')
        plt.title("Proporcion de infectados a lo largo de 100u de tiempo.")
        plt.legend()
        plt.show()
        #plt.savefig('Figuras/simulacion1.png')


ejec = Simulacion7()
ejec.main()
```

B.2. Funcionalidades del núcleo

B.2.1. FuncionesNucleo.py

```
import random
from igraph import *
import pylab


class FuncionesNucleo():

    def generaGrafo(self, tipo, n=10000, conectado=True, p=0.25):
        if tipo == "Erdos_Renyi":
            # Erdos_Renyi(n, p, m, directed=False, loops=False)
            # n = n vertices, p = probabilidad de aristas
            # (si hay p, no hay m)
            # m = numero de aristas
            grafo = Graph.Erdos_Renyi(n, m=2 * n)
        elif tipo == "Lattice":
            # Lattice(dim, nei=1, directed=False, mutual=True, circular=True)
            # dim = lista con las diemnsiones del grafo(sqrst(n) * sqrt(n)
            # en este caso)
            # nei = distancia a la que dos vertices estan conectados
            dim = [int(n ** 0.5), int(n ** 0.5)]
            grafo = Graph.Lattice(dim)
        elif tipo == "Grafo_Aleatorio":
            #
            r = 2 / ((n - 1) * pi) ** 0.5
            grafo = Graph.GRG(n, r)
        elif tipo == "Scale_Free":
            # Barabasi(n, m, outpref=False, directed=False, power=1,
            # zero_appeal=1, implementation="psumtree", start_from=None)
            # n = n vertices, m = numero de aristas de saldia en cada nodo,
            # o una lista con todas las aristas power = constante de
            # potencia, sino se especifica linear
            grafo = Graph.Barabasi(n, 2, power=2.2)
        elif tipo == "Small_Worlds":
            # Watts_Strogatz(dim, size, nei, p, loops=False, multiple=False)
            # Watts_Strogats parte de un grafo tipo Laticce
            # dim = dimensiones del lattice, size = tamanyo de todas
            # las dimensiones
            # nei = distancia a la que dos vertices estan conectados
            # p = rewiring probability
            grafo = Graph.Watts_Strogatz(1, n, 2, p)
        else:
            raise ValueError("No soportamos ese tipo de grafo.")

        # Comprobamos que el grafo este conectado, sino lo conectamos
        # nosotros
        if conectado and not grafo.is_connected():
            grafo = grafo.clusters().giant()
```

```
        return grafo

    def simulacion(self, modelo, tipo_grafo, tam_grafo=10000, n_infectados=1,
                   susceptibles=10000, runs=10, time=100, *args, **kwds):

        results = []
        aux = [] # variable para CdE
        # los susceptibles estan por defecto con 10000 para igualar
        # el tamanyo del grafo por defecto. Por tanto
        if susceptibles == 10000:
            susceptibles = tam_grafo
        for i in xrange(runs):
            current_run = []
            aux2 = []

            ## Inicializamos grafo y modelo
            grafo = self.generaGrafo(tipo_grafo, n=tam_grafo)
            model = modelo(grafo, *args, **kwds)

            if susceptibles != tam_grafo:
                s = random.sample(range(grafo.vcount()), int(susceptibles%1))
                model.compartimentos.move_vertices(s, "S")
            else:
                for v in grafo.vs:
                    model.compartimentos.move_vertice(v.index, "S")
            ## Infectamos el numero de nodos indicado
            infectados = random.sample(range(grafo.vcount()), n_infectados)
            model.compartimentos.move_vertices(infectados, "I")

            ## realizamos los pasos
            for t in xrange(time):
                model.step()
                p_infectados = model.compartimentos.tam_relativo("I")
                current_run.append(p_infectados)
                if t < 10:
                    aux2.append(p_infectados)

            results.append(current_run)
            aux.append(aux2)
        return results, aux

    def simulacionHub(self, modelo, tipo_grafo, tam_grafo=10000, num_infectados=
                      susceptibles=10000, runs=10, time=100, *args, **kwds):
        """ Simulacion"""

        results = []
        aux = [] # variable para CdE
        # los susceptibles estan por defecto con 10000 para igualar
        # el tamanyo del grafo por defecto. Por tanto
        if susceptibles == 10000:
            susceptibles = tam_grafo
```

```
    for i in xrange(runs):
        current_run = []
        aux2 = []
        n_infectados = num_infectados

        ## Inicializamos grafo y modelo
        grafo = self.generaGrafo(tipo_grafo, n=tam_grafo)
        model = modelo(grafo, *args, **kwds)

        if susceptibles != tam_grafo:
            s = random.sample(range(grafo.vcount()), int(susceptibles%1))
            model.compartimentos.move_vertices(s, "S")
        else:
            for v in grafo.vs:
                model.compartimentos.move_vertice(v.index, "S")

        # Infectamos el numero de nodos indicado de forma que sean
        # los de mayor grado posible
        hubs = []
        infectados = []

        j = 0
        while n_infectados > len(hubs):
            for v in grafo.vs.select(_degree=(grafo.maxdegree() - j)):
                hubs.append(v.index)
            if len(hubs) == 1:
                infectados.append(hubs[0])
            else:
                if len(hubs) > n_infectados:
                    infectados = infectados + random.sample(hubs,
                                                            n_infectados)
                else:
                    infectados = infectados + random.sample(hubs, len(hubs))
            n_infectados = n_infectados - len(hubs)
            j += 1
        model.compartimentos.move_vertices(infectados, "I")

        # realizamos los pasos
        for t in xrange(time):
            model.step()
            p_infectados = model.compartimentos.tam_relativo("I")
            current_run.append(p_infectados)
            if t < 20:
                aux2.append(p_infectados)


        results.append(current_run)
        aux.append(aux2)
    return results, aux
```

```
    def simulacionAntiHub(self, modelo, tipo_grafo, tam_grafo=10000,
                          n_infectados=1, susceptibles=10000, runs=10, time=100,
                          *args, **kwds):
        """ Simulacion"""

        results = []
        # los susceptibles estan por defecto con 10000 para igualar
        # el tamanyo del grafo por defecto. Por tanto
        if susceptibles == 10000:
            susceptibles = tam_grafo

        for i in xrange(runs):
            print "Ronda %d" % i
            current_run = []

            ## Inicializamos grafo y modelo
            grafo = self.generaGrafo(tipo_grafo, n=tam_grafo)
            model = modelo(grafo, *args, **kwds)

            if susceptibles != tam_grafo:
                s = random.sample(range(grafo.vcount()), susceptibles)
                model.compartimentos.move_vertices(s, "S")
            else:
                for v in grafo.vs:
                    model.compartimentos.move_vertice(v.index, "S")

            ## Infectamos el numero de nodos indicado
            hubs = []
            infectados = []

            j = 1
            while n_infectados > len(hubs):
                for v in grafo.vs.select(_degree=j):
                    hubs.append(v.index)
                if len(hubs) == 1:
                    infectados.append(hubs[0])
                else:
                    if len(hubs) > n_infectados:
                        infectados = infectados + random.sample(hubs,
                                                                n_infectados)
                    else:
                        infectados = infectados + random.sample(hubs, len(hubs))
                n_infectados = n_infectados - len(hubs)
                j += 1

            model.compartimentos.move_vertices(infectados, "I")

            # realizamos los pasos
            for t in xrange(time):
                model.step()
                p_infectados = model.compartimentos.tam_relativo("I")
```

```
            current_run.append(p_infectados)

        results.append(current_run)
    return results

def simulacionInteligente(self, modelo, tipo_grafo, tam_grafo=10000,
                          n_infectados=1, susceptibles=10000, runs=10,
                          time=100, *args, **kwds):
    results = []
    aux = []
    for i in xrange(runs):
        current_run = []
        aux2 = []

        ## Inicializamos grafo y modelo
        grafo = self.generaGrafo(tipo_grafo, n=tam_grafo)
        model = modelo(grafo, *args, **kwds)

        if susceptibles != tam_grafo:
            s = random.sample(range(grafo.vcount()), int(susceptibles))
            model.compartimentos.move_vertices(s, "S")
        else:
            for v in grafo.vs:
                model.compartimentos.move_vertice(v.index, "S")
        ## Infectamos el numero de nodos indicado
        infectados = random.sample(range(grafo.vcount()), n_infectados)
        for i in infectados:
            model.compartimentos.move_vertice(i, "I")

        ## realizamos los pasos
        for t in xrange(time):
            if t <= 5:
                model.stepInteligente()
            else:
                model.step()
            p_infectados = model.compartimentos.tam_relativo("I")
            current_run.append(p_infectados)
            if t < 20:
                aux2.append(p_infectados)

        results.append(current_run)
        aux.append(aux2)
    return results, aux

def simulacionInteligente2(self, modelo, tipo_grafo, tam_grafo=10000,
                           n_infectados=1, susceptibles=10000, runs=10,
                           time=100, beta=0.3, *args, **kwds):
    results = []
    if susceptibles == 10000:
        susceptibles = tam_grafo
    for i in xrange(runs):
```

```
            current_run = []

            ## Inicializamos grafo y modelo
            grafo = self.generaGrafo(tipo_grafo, n=tam_grafo)
            model = modelo(grafo, *args, **kwds)

            if susceptibles != tam_grafo:
                s = random.sample(range(grafo.vcount()), susceptibles)
                model.compartimentos.move_vertices(s, "S")
            else:
                for v in grafo.vs:
                    model.compartimentos.move_vertice(v.index, "S")
            ## Infectamos el numero de nodos indicado
            infectados = random.sample(range(grafo.vcount()), n_infectados)
            for i in infectados:
                # Comprobamos que son susceptibles
                if model.compartimentos.get_state(i) == "S":
                    model.compartimentos.move_vertice(i, "I")

            ## realizamos los pasos
            for t in xrange(time):
                model.stepInteligente2(beta)
                p_infectados = model.compartimentos.tam_relativo("I")
                current_run.append(p_infectados)

            results.append(current_run)
        return results

    def simulacionColores(self, modelo, tipo_grafo, tam_grafo=10000,
                          n_infectados=1, susceptibles=10000, runs=10,
                          time=100, beta=0.3, *args, **kwds):
        results = []
        if susceptibles == 10000:
            susceptibles = tam_grafo
        for i in xrange(runs):
            current_run = []

            ## Inicializamos grafo y modelo
            grafo = self.generaGrafo(tipo_grafo, n=tam_grafo)
            model = modelo(grafo, *args, **kwds)

            ## Infectamos el numero de nodos indicado
            if susceptibles != tam_grafo:
                s = random.sample(range(grafo.vcount()), int(susceptibles))
                model.compartimentos.move_vertices(s, "S")
            for id in model.compartimentos.compartimentos['S']:
                grafo.vs[id]['color'] = 'green'
            ## Infectamos el numero de nodos indicado
            infectados = random.sample(range(grafo.vcount()), n_infectados)
            for i in infectados:
                model.compartimentos.move_vertice(i, "I")
```

```
            ## Marcamos los pacientes cero en negro
            for id in model.compartimentos.compartimentos['I']:
                grafo.vs[id]['color'] = 'black'

            plot(grafo, "simulacion5_estadoinicial.png")
            ## realizamos los pasos
            for t in xrange(7):
                model.step()
                if (t %2) == 0:
                    for id in model.compartimentos.compartimentos['X']:
                        if grafo.vs[id]['color'] != 'black':
                            grafo.vs[id]['color'] = 'grey'
                        else:
                            grafo.vs[id]['color'] = 'cyan'
                    for id in model.compartimentos.compartimentos['S']:
                        grafo.vs[id]['color'] = 'yellow'
                    for id in model.compartimentos.compartimentos['I']:
                        if grafo.vs[id]['color'] != 'black':
                            grafo.vs[id]['color'] = 'red'
                    for id in model.compartimentos.compartimentos['R']:
                        grafo.vs[id]['color'] = 'green'
                    fichero = "Figuras/simulacion5_" + "%d" %t + ".png"
                    plot(grafo, fichero)

            model.beta = 0.7
            model.gamma = 0.5

            ## realizamos los pasos
            for t in xrange(7,15):
                model.step()
                if (t %2) == 0:
                    for id in model.compartimentos.compartimentos['X']:
                        if grafo.vs[id]['color'] != 'black':
                            grafo.vs[id]['color'] = 'grey'
                        else:
                            grafo.vs[id]['color'] = 'cyan'
                    for id in model.compartimentos.compartimentos['S']:
                        grafo.vs[id]['color'] = 'yellow'
                    for id in model.compartimentos.compartimentos['I']:
                        if grafo.vs[id]['color'] != 'black':
                            grafo.vs[id]['color'] = 'red'
                    for id in model.compartimentos.compartimentos['R']:
                        grafo.vs[id]['color'] = 'green'
                    fichero = "Figuras/simulacion5_" + "%d" %t + ".png"
                    plot(grafo, fichero)

        return results

    def simulacionGrupoInicial(self, modelo, tipo_grafo, tam_grafo=10000,
                               n_infectados=1, susceptibles=10000, runs=10,
```

```
                                time=100, *args, **kwds):
        """ Funcion simulacion adaptada, para que el grupo inicial de
        infectados sean vecinos. """

        results = []
        aux = []
        # los susceptibles estan por defecto con 10000 para igualar
        # el tamanyo del grafo por defecto. Por tanto
        if susceptibles == 10000:
            susceptibles = tam_grafo
        for i in xrange(runs-1):
            current_run = []
            aux2 = []

            # Inicializamos grafo y modelo
            grafo = self.generaGrafo(tipo_grafo, n=tam_grafo)
            model = modelo(grafo, *args, **kwds)

            if susceptibles != tam_grafo:
                s = random.sample(range(grafo.vcount()), int(susceptibles))
                model.compartimentos.move_vertices(s, "S")

            # Infectamos el numero de nodos indicado
            infectados = []
            paciente_cero = random.sample(range(grafo.vcount()), 1)[0]
            infectados.append(paciente_cero)
            n_infectados2 = n_infectados -1
            while n_infectados2 > 0:
                neis = grafo.neighbors(paciente_cero)
                for nei in neis:
                    if n_infectados2 > 0:
                        infectados.append(nei)
                        n_infectados2 -= 1

            model.compartimentos.move_vertices(infectados, "I")

            ## realizamos los pasos
            for t in xrange(time):
                model.step()
                p_infectados = model.compartimentos.tam_relativo("I")
                current_run.append(p_infectados)
                if t < 20:
                    aux2.append(p_infectados)

            results.append(current_run)
            aux.append(aux2)
        return results, aux


    def plot_results(self, results, color):
        for r in results:
```

```
            print r
            ## Calculate the means for each time point
            averaged = [mean(items) for items in izip(*r)]

            ## Create the argument list for pylab.plot
            args = []
            for row in r:
                args += [row, 'k-']
            args += [averaged, color]

            ## Creathe the plot
            pylab.plot(*args)

    def hallarUmbralCritico(self, modelo, tipo, betas, *args, **kwds):
        for beta in betas:
            results = self.simulacion(modelo, tipo, 100, beta=beta, *args,
                                      **kwds)
            self.plot_results(results, 'r-o')
            pylab.show()
            media = mean(run[-1] for run in results)
            print("Beta: %.2f  Media: %.4f" %(beta, media))
```

B.2.2. FuncionesAuxiliares.py

```
import random

def muestraAleatoria(nodos, p):
    """Funcion que permite seleccionar unos nodos de forma
    aleatorio para simular la transicion de estados en los
    distintos modelos, para ello usa la libreria random"""
    return [nodo for nodo in nodos if random.random() < p]
```

B.3. Ficheros del modelo compartimental

B.3.1. Compartimentos.py

```
""" Interfaz para los compartimentos del modelo"""
class Compartimentos(object):
    def __init__(self, grafo, comps):
        """ Mediante un grafo y los codigos de los compartimentos, comps,
        crea los compartimentos del modelo."""
        self.comps = list(comps)
        self.n = grafo.vcount()

        first_comp = self.comps[0]
        self.states = [first_comp] * self.n

        self.compartimentos = dict()
        for code in comps:
            self.compartimentos[code] = set()
        self.compartimentos[first_comp].update(xrange(self.n))
```

```
    def __getitem__(self, comp):
        """Devuelve el compartimento asociado
        al codigo comp."""

        return self.compartimentos[comp]

    def get_state(self, vertice):
        """Devuelve el codigo del compartimento
        en el que se encuentra el vertice."""
        return self.states[vertice]

    def move_vertice(self, vertice, comp):
        """Cambia el vertice de compartimento """
        self.compartimentos[self.states[vertice]].remove(vertice)
        self.states[vertice] = comp
        self.compartimentos[comp].add(vertice)

    def move_vertices(self, vertices, code):
        """ Llama a move_vertice para cada vertice """
        for vertex in vertices:
            self.move_vertice(vertex, code)

    def tam_relativo(self, code):
        """ Devuelve el tamanyo relativo del compartimento
        code. """
        return len(self.compartimentos[code]) / float(self.n)
```

B.3.2. ModeloCompartimental.py

```
from Compartimentos import Compartimentos

class ModeloCompartimental(object):
    """Interfaz para la clase de los modelos
    epidemiologicos compartimentales."""

    def __init__(self, grafo, codes):
        """Inicializa un.modelo compartimental sobre un
        grafo que recibe como argumento y el codigo
        asociado a los compartimentos"""
        self.grafo = grafo
        self.compartimentos = Compartimentos(grafo, codes)
        self.reset()

    def tam_relativo_compartimentos(self):
        """Returns the relative sizes of each comparment in the
        model."""
        return [self.compartimentos.tam_relativo(cmp)
                for cmp in self.compartimentos.cmps]

    def reset(self):
```

```
        """Resets the compartments to an initial state. This
        method must be overriden in subclasses."""
        raise NotImplementedError

    def step(self):
        """ Implements the logic of the epidemic model. This method
        must be overriden by subclasses."""
        raise NotImplementedError

    def step_many(self, n):
        """Runs 'n' steps of the epidemic model at once by
        calling 'step' multiple times."""
        for i in xrange(n):
            self.step()
```

B.3.3. ModeloSIS.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-

from ModeloCompartimental import ModeloCompartimental
from FuncionesAuxiliares import muestraAleatoria
from igraph import mean
import random as random


class ModeloSIS(ModeloCompartimental):
    """SIS, modelo epidemiologico para redes"""

    def __init__(self, g, beta=0.1, gamma=0.2):
        """Inicializamos los compartimentos y damos valores a los parametros"""
        ModeloCompartimental.__init__(self, g, "SI")
        self.beta = float(beta)
        self.gamma = float(gamma)

    def reset(self):
        """Inicializamos toda la poblacion a susceptibles"""
        vs = xrange(self.grafo.vcount())
        self.compartimentos.move_vertices(vs, "S")

    def step(self):
        """ Un paso del modelo SIS"""
        ## Se extiende la infeccion
        s_to_i = set()  # Inicializamos un conjunto para los traspasos

        ## Calculamos para cada vertice infectado
        ## Cuales de sus vecinos seran infectados
        for v in self.compartimentos["I"]:
            neis = self.grafo.neighbors(v)
            s_to_i.update(muestraAleatoria(neis, self.beta))
            ## Aplicamos los cambios
        self.compartimentos.move_vertices(s_to_i, "I")
```

```
        ## Algunos de los infectados se curaran
        i_to_s = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_s, "S")

    def stepInteligente(self):
        """ Un paso del modelo SIS en el que solo ataca nodos
        con un grado superior a la media."""
        ## Se extiende la infeccion
        s_to_i = set()  # Inicializamos un conjunto para los traspasos

        ## Grado medio del grafo
        g_medio = mean(self.grafo.degree()) // 1

        ## Calculamos para cada vertice infectado
        ## Cuales de sus vecinos seran infectados
        for v in self.compartimentos["I"]:
            neis = self.grafo.neighbors(v)
            s_to_i.update([nodo for nodo in neis if
                           self.grafo.vs[nodo].degree() > g_medio])
            ## Aplicamos los cambios
        self.compartimentos.move_vertices(s_to_i, "I")

        ## Algunos de los infectados se curaran
        i_to_s = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_s, "S")

    def stepInteligente2(self, beta):
        """ Un paso del modelo SIS en el que solo ataca nodos
        con un grado superior a la media."""
        ## Se extiende la infeccion
        s_to_i = set()  # Inicializamos un conjunto para los traspasos

        ## Grado medio del grafo
        g_medio = mean(self.grafo.degree()) // 1

        ## Calculamos para cada vertice infectado
        ## Cuales de sus vecinos seran infectados
        for v in self.compartimentos["I"]:
            neis = self.grafo.neighbors(v)
            s_to_i.update([nodo for nodo in neis if
                           ((self.grafo.vs[nodo].degree() < g_medio) and
                            (random.random() < beta))])
            ## Aplicamos los cambios
        self.compartimentos.move_vertices(s_to_i, "I")

        ## Algunos de los infectados se curaran
        i_to_s = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_s, "S")
```

B.3.4. ModeloSIR.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
from ModeloCompartimental import ModeloCompartimental
from FuncionesAuxiliares import muestraAleatoria
from igraph import mean
import random as random


class ModeloSIR(ModeloCompartimental):
    """ Modelo epidemiologico SIR para redes """

    def __init__(self, graph, beta=0.1, gamma=0.2):
        """ Construye el modelo compartimental sobre el
        grafo que recibe como argumento, y le asigna los
        valores de transicion. """
        ModeloCompartimental.__init__(self, graph, "SIR")
        self.beta = float(beta)
        self.gamma = float(gamma)

    def reset(self):
        """ Inicializamos toda la poblacion a susceptibles """
        vs = xrange(self.grafo.vcount())
        self.compartimentos.move_vertices(vs, "S")

    def step(self):
        """" Un paso del modelo SIR"""
        ## Extendemos la infeccion desde los nodos infectados
        for v in self.compartimentos["I"].copy():
            neis = self.grafo.neighbors(v)
            ## Algunos nodos se infectan
            for nei in muestraAleatoria(neis, self.beta):
                if self.compartimentos.get_state(nei) == "S":
                    self.compartimentos.move_vertice(nei, "I")

        ## Algunos se recuperan
        i_to_r = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_r, "R")

    def stepInteligente(self):
        """ Un paso del modelo SIS en el que solo ataca nodos
        con un grado superior a la media."""
        ## Se extiende la infeccion
        s_to_i = set()  # Inicializamos un conjunto para los traspasos

        ## Grado medio del grafo
        g_alto = mean(self.grafo.degree()) * 1.8 // 1

        ## Calculamos para cada vertice infectado
        ## Cuales de sus vecinos seran infectados
        for v in self.compartimentos["I"]:
            neis = self.grafo.neighbors(v)
```

```
            s_to_i.update([nodo for nodo in neis if
                           self.grafo.vs[nodo].degree() > g_alto])
            ## Aplicamos los cambios
        self.compartimentos.move_vertices(s_to_i, "I")

        ## Algunos se recuperan
        i_to_r = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_r, "R")

    def stepInteligente2(self, beta):
        """ Un paso del modelo SIR en el que solo ataca nodos
        con un grado superior a la media."""
        ## Se extiende la infeccion
        s_to_i = set()  # Inicializamos un conjunto para los traspasos

        ## Grado medio del grafo
        g_alto = mean(self.grafo.degree()) * 1.8 // 1

        ## Calculamos para cada vertice infectado
        ## Cuales de sus vecinos seran infectados
        for v in self.compartimentos["I"]:
            neis = self.grafo.neighbors(v)
            s_to_i.update([nodo for nodo in neis if
                           ((self.grafo.vs[nodo].degree() < g_alto) and
                            (random.random() < beta))])
            ## Aplicamos los cambios
        self.compartimentos.move_vertices(s_to_i, "I")

        ## Algunos se recuperan
        i_to_r = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_r, "R")
```

B.3.5. ModeloSIRS.py

```
from ModeloSIR import ModeloSIR
from FuncionesAuxiliares import muestraAleatoria

class ModeloSIRS(ModeloSIR):
    """Modelo epidemiologico SIRS para redes. Esta clase
    extiende la del modelo SIR"""
    def __init__(self, graph, beta=0.1, gamma=0.2, lambda_=0.4):
        ModeloSIR.__init__(self, graph, beta, gamma)
        ## El nuevo valor de transicion entre
        ## recuperados y susceptibles
        self.lambda_ = float(lambda_)


    def step(self):
        """Respecto al SIR solo incluye un paso mas donde los
        recuperados pueden pasar a ser susceptibles."""
        r_to_s = muestraAleatoria(self.compartimentos["R"], self.lambda_)
        self.compartimentos.move_vertices(r_to_s, "S")
```

```
        ## Ahora llamamos al SIR
        ModeloSIR.step(self)
```

B.3.6. ModeloXSIR.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
from ModeloCompartimental import ModeloCompartimental
from FuncionesAuxiliares import muestraAleatoria
from igraph import mean
import random as random


class ModeloXSIR(ModeloCompartimental):
    """ Modelo epidemiologico SIR para redes, con un nuevo estado
     de inmunes que llamaremos X. """

    def __init__(self, graph, beta=0.1, gamma=0.2):
        """ Construye el modelo compartimental sobre el
        grafo que recibe como argumento, y le asigna los
        valores de transicion. """
        ModeloCompartimental.__init__(self, graph, "XSIR")
        self.beta = float(beta)
        self.gamma = float(gamma)

    def reset(self):
        """ Inicializamos toda la poblacion a susceptibles """
        vs = xrange(self.grafo.vcount())
        self.compartimentos.move_vertices(vs, "X")

    def step(self):
        """" Un paso del modelo XSIR que es lo mismo que de SIR"""
        ## Extendemos la infeccion desde los nodos infectados
        for v in self.compartimentos["I"].copy():
            neis = self.grafo.neighbors(v)
            ## Algunos nodos se infectan
            for nei in muestraAleatoria(neis, self.beta):
                if self.compartimentos.get_state(nei) == "S":
                    self.compartimentos.move_vertice(nei, "I")

        ## Algunos se recuperan
        i_to_r = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_r, "R")

    def stepInteligente(self):
        """ Un paso del modelo SIS en el que solo ataca nodos
        con un grado superior a la media."""
        ## Se extiende la infeccion
        s_to_i = set()  # Inicializamos un conjunto para los traspasos

        # Percentil 90 de la dist del grado
        # de los nodos
```

```
        g_alto = mean(self.grafo.degree())  // 1
        ## Calculamos para cada vertice infectado
        ## Cuales de sus vecinos seran infectados
        for v in self.compartimentos["I"]:
            neis = self.grafo.neighbors(v)
            for nodo in neis:
                aux = []
                if (self.grafo.vs[
                    nodo].degree() > g_alto and self.compartimentos.get_state(
                        nodo) == 'S'):
                    aux.append(nodo)
            s_to_i.update(aux)
            # Ahora calculamos los infectados por el proceso
            # normal
            aux = []
            for nei in muestraAleatoria(neis, self.beta):
                if self.compartimentos.get_state(nei) == "S":
                    aux.append(nei)
            s_to_i.update(aux)


            # s_to_i.update([nodo for nodo in neis if
            #                  self.grafo.vs[nodo].degree() > g_alto
            #                 and self.compartimentos.get_state(nodo) == 'S'])
            ## Aplicamos los cambios
        self.compartimentos.move_vertices(s_to_i, "I")

        ## Algunos se recuperan
        i_to_r = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_r, "R")

    def stepInteligente2(self, beta):
        """ Un paso del modelo SIS en el que solo ataca nodos
        con un grado superior a la media."""
        ## Se extiende la infeccion
        s_to_i = set()  # Inicializamos un conjunto para los traspasos

        ## Grado medio del grafo
        g_medio = mean(self.grafo.degree()) // 1

        ## Calculamos para cada vertice infectado
        ## Cuales de sus vecinos seran infectados
        for v in self.compartimentos["I"]:
            neis = self.grafo.neighbors(v)
            s_to_i.update([nodo for nodo in neis if
                           ((self.grafo.vs[nodo].degree() < g_medio) and
                            (random.random() < beta))])
            ## Aplicamos los cambios
        self.compartimentos.move_vertices(s_to_i, "I")

        ## Algunos se recuperan
```

```
        i_to_r = muestraAleatoria(self.compartimentos["I"], self.gamma)
        self.compartimentos.move_vertices(i_to_r, "R")
```

B.4. Código de los ejemplos

B.4.1. generaGrafoSW.py

Código correspondiente a la figura 2.8.

```
# #!/usr/bin/python
# # -*- coding: latin-1 -*-
import cairo
from igraph.drawing.text import TextDrawer
from igraph import *
from FuncionesNucleo import generaGrafo

g1 = generaGrafo("Small_Worlds", n=200)

# Asignamos colores
for v in VertexSeq(g1):
    if v.degree() > 9:
        v['color'] = 'cyan'
        v['size'] = 50
    elif v.degree() > 8:
        v['color'] = 'yellow'
        v['size'] = 40
    elif v.degree() > 7:
        v['color'] = 'green'
        v['size'] = 30
    elif v.degree() > 6:
        v['color'] = 'pink'
        v['size'] = 20

# Construimos el plot
plot = Plot("smallworld.png", bbox=(600, 400), background="white")
plot.add(g1, bbox=(20, 80, 500, 300))
# Añadimos el grafo al plot
plot.redraw()
# Preparamos el canvas donde escribiremos el grafo
ctx = cairo.Context(plot.surface)
ctx.set_font_size(14)
## Dibujamos la leyenda
drawer = TextDrawer(ctx,"Cyan : Degree(v)>9", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 20, width=580)
drawer = TextDrawer(ctx,"Amarillo : Degree(v)>8", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 40, width=580)
drawer = TextDrawer(ctx,"Verde : Degree(v)>7", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 60, width=580)
drawer = TextDrawer(ctx,"Rosa : Degree(v)>6", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 80, width=580)
drawer = TextDrawer(ctx,"Rojo : Degree(v)<=5", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 100, width=580)
```

```
# Save the plot
plot.save()
plot.show()
```

B.4.2. generaGrafoSF.py

Código correspondiente a la figura 2.11.

```
# #!/usr/bin/python
# # -*- coding: latin-1 -*-
import cairo
from igraph.drawing.text import TextDrawer
from igraph import *
from FuncionesNucleo import generaGrafo

g1 = generaGrafo("Scale_Free", n = 200)

# Asignamos colores
for v in VertexSeq(g1):
    if v.degree() > 9:
        v['color'] = 'cyan'
        v['size'] = 50
    elif v.degree() > 8:
        v['color'] = 'yellow'
        v['size'] = 40
    elif v.degree() > 7:
        v['color'] = 'green'
        v['size'] = 30
    elif v.degree() > 6:
        v['color'] = 'pink'
        v['size'] = 20

# Construimos el plot
plot = Plot("scalefree.png", bbox=(600, 400), background="white")
plot.add(g1, bbox=(20, 80, 500, 300))
# Añadimos el grafo al plot
plot.redraw()
# Preparamos el canvas donde escribiremos el grafo
ctx = cairo.Context(plot.surface)
ctx.set_font_size(14)
## Dibujamos la leyenda
drawer = TextDrawer(ctx,"Cyan : Degree(v)>9", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 20, width=580)
drawer = TextDrawer(ctx,"Amarillo : Degree(v)>8", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 40, width=580)
drawer = TextDrawer(ctx,"Verde : Degree(v)>7", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 60, width=580)
drawer = TextDrawer(ctx,"Rosa : Degree(v)>6", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 80, width=580)
```

```
drawer = TextDrawer(ctx,"Rojo : Degree(v)<=5", halign=TextDrawer.RIGHT)
drawer.draw_at(0, 100, width=580)


# Save the plot
plot.save()
plot.show()
```

B.4.3. pforSW.py

Código correspondiente a la figura 2.9.

```
# #!/usr/bin/python
# # -*- coding: latin-1 -*-
from igraph import *
import random
import numpy as np


class pforSW(object):

    def main(self):
        ficheros = ["imagen1SW_p.png", "imagen2SW_p.png",
                                       "imagen3SW_p.png"]
        probs = [0, 0.1, 1]
        i = 0
        for f in ficheros:
            self.generaGrafo(f, probs[i])
            i += 1


    def rondaWatts(self, g1, p):
        for v in VertexSeq(g1):
            ## Tenemos una probabilidad de 0.5 de recablear
            if random.random() < p:
                ## Esta parte se encarga de escoger un eje
                ## aleatorio del vertice
                ## random.choice(g1.es.select(v.index))

                g1.delete_edges(random.choice(g1.es.select(v.index)).index)
                ## Escogemos el vertice al que enlazar
                aleatorio = int(round(random.uniform(0, 19), 0))
                while aleatorio == v.index:
                    aleatorio = int(round(random.uniform(0, 20), 0))
                g1.add_edge(v.index, aleatorio)

    def generaGrafo(self, fichero, p):
        ## Creamos grafo de anillo base.
        g1 = Graph.Ring(50)

        ## Tenemos 20 nodos, ahora pongamosle nombres para trabajar
```

```
        ## con ellos
        i = 1
        for v in VertexSeq(g1):
            v['name'] = "%d" % i
            i += 1

        ## Ahora añadimos una arista que enlace cada vertice
        ## con el vecino de su vecino en el sentido horario
        i = 1
        while i < 49:
            g1.add_edge("%d" %i, "%d" %(i + 2))
            i += 1
        ## añadimos los dos ultimos ejes
        g1.add_edge('49', '1')
        g1.add_edge('50', '2')


        ## Aplicamos el proceso de rewire con probabilidad p
        ## 3 veces.
        i = 0
        while i < 3:
            self.rondaWatts(g1, p)
            i += 1

        ## Guardamos el grafo
        plot(g1, fichero)

eje = pforSW()
eje.main()
```

B.4.4. algoritmoWatts.py

Código correspondiente a la Figura 2.10.

```
# #!/usr/bin/python
# # -*- coding: latin-1 -*-
from igraph import *
import random
import numpy as np

class algoritmoWatts(object):

    def main(self):

        ## Creamos grafo de anillo base.
        g1 = Graph.Ring(50)

        ## Tenemos 20 nodos, ahora pongamosle nombres para trabajar
        ## con ellos
        i = 1
        for v in VertexSeq(g1):
```

```
        v['name'] = "%d" %i
        i += 1

    ## Ahora añadimos una arista que enlace cada vertice
    ## con el vecino de su vecino en el sentido horario
    i = 1
    while i < 49:
        g1.add_edge("%d" %i, "%d" %(i + 2))
        i += 1
    ## añadimos los dos ultimos ejes
    g1.add_edge('49', '1')
    g1.add_edge('50', '2')

    layout = g1.layout_circle
    plot(g1, "imagen1sw.png", vertex_label=g1.degree())
    # plot.show()

    ## Aplicamos el proceso de rewire con probabilidad p
    for v in VertexSeq(g1):
        ## Tenemos una probabilidad de 0.5 de recablear
        if random.random() < 0.5:
            ## Esta parte se encarga de escoger un eje
            ## aleatorio del vertice
            ## random.choice(g1.es.select(v.index))

            g1.delete_edges(random.choice(g1.es.select(v.index)).index)
            ## Escogemos el vertice al que enlazar
            aleatorio = int(round(random.uniform(0, 19), 0))
            while aleatorio == v.index:
                aleatorio = int(round(random.uniform(0, 20), 0))
            g1.add_edge(v.index, aleatorio)

    self.ajustarTamanyo(g1)
    plot(g1, "imagen2sw.png", vertex_label=g1.degree())

    ## Aplicamos otra ronda
    for v in VertexSeq(g1):
        ## Tenemos una probabilidad de 0.5 de recablear
        if random.random() < 0.5:
            ## Esta parte se encarga de escoger un eje
            ## aleatorio del vertice
            ## random.choice(g1.es.select(v.index))

            g1.delete_edges(random.choice(g1.es.select(v.index)).index)
            ## Escogemos el vertice al que enlazar
            aleatorio = int(round(random.uniform(0, 19), 0))
            while aleatorio == v.index:
                aleatorio = int(round(random.uniform(0, 20), 0))
            g1.add_edge(v.index, aleatorio)

    self.ajustarTamanyo(g1)
```

```
        plot(g1, "imagen3sw.png", vertex_label=g1.degree())

        plot(g1, vertex_label=g1.degree())

    def ajustarTamanyo(self, grafo):
        grados = sorted(grafo.degree())
        mayor = round(np.percentile(grados, 75))
        media = round(np.percentile(grados, 50))
        menor = round(np.percentile(grados, 25))

        for v in VertexSeq(grafo):
            if v.degree() > mayor:
                v['color'] = 'cyan'
                v['size'] = 60
            elif v.degree() > media:
                v['color'] = 'yellow'
                v['size'] = 40
            elif v.degree() > menor:
                v['color'] = 'green'
                v['size'] = 25
            else:
                v['color'] = 'red'
                v['size'] = 15

## Ejecutamos el algoritmo
ejec = algoritmoWatts()
ejec.main()
```

B.4.5. algoritmoBarabasi.py

Código correspondiente a la figura 2.12.

```
# #!/usr/bin/python
# # -*- coding: latin-1 -*-
from igraph import *
import numpy as np

class algoritmoBarabasi(object):

    def main(self):
        ## Vamos a ejemplificar como nace una
        ## red scale free. Para ello supondremos
        ## que los nodos con grado superior al grado
        ## medio tienen una probabilidad d
        g1 = Graph()
        i = 0
        j = 0
        nombres_fichero = ["imagen1sf.png", "imagen2sf.png", "imagen3sf.png",
                           "imagen4sf.png", "imagen5sf.png", "imagen6sf.png"]
```

```
        g1.add_vertex()
        g1.add_vertex()
        g1.add_edge(g1.vs[0], g1.vs[1])
        print g1.vs[0].degree()
        print g1.vs[1].degree()

        while i < 30:
            g1.add_vertex()
            self.preferalAttachment(g1, g1.vs[i + 2])
            self.preferalAttachment(g1, g1.vs[i + 2])
            if (i %5) == 0:
                if i != 0:
                    self.ajustarTamanyo(g1)
                plot(g1, nombres_fichero[j], vertex_label=g1.degree())
                j += 1
            i += 1

        plot(g1, vertex_label=g1.degree())

    def preferalAttachment(self, grafo, nodo):
        fin = 0

        ## Empezamos calculando los cuatro percentiles
        grados = grafo.degree()
        sum_grados = np.sum(grados)

        ## Preparamos la distribucion de probabilidad
        dist_prob = []
        for v in grafo.vs:
            dist_prob.append(v.degree()/float(sum_grados))


        ## Escogemos un nodo aleatorio distinto del nuevo
        aleat = np.random.choice(grafo.vs, p=dist_prob)
        while fin != 1:
            ## Evitamos los bucles
            if nodo == aleat:
                aleat = np.random.choice(grafo.vs)
                break

            ## Comprobamos que no hay ejes repetidos
            ejes1 = grafo.es.select(_source=aleat.index)
            ejes2 = grafo.es.select(_source=aleat.index)
            flag = 0
            for e in ejes1:
                flag = e in grafo.es.select(_source=nodo.index)
                if flag:
                    break
            if flag:
                continue
            else:
```

```
                fin = 1


        ## Añadimos el eje
        grafo.add_edge(nodo, aleat)


    def ajustarTamanyo(self, grafo):
        grados = sorted(grafo.degree())
        mayor = round(np.percentile(grados, 75))
        media = round(np.percentile(grados, 50))
        menor = round(np.percentile(grados, 25))

        for v in VertexSeq(grafo):
            if v.degree() > mayor:
                v['size'] = 60
            elif v.degree() > media:
                v['size'] = 40
            elif v.degree() > menor:
                v['size'] = 25
            else:
                v['size'] = 15

## Ejecutamos el algoritmo
ejec = algoritmoBarabasi()
ejec.main()
```